gitprint

Branch	main
Commit	699fff0
Author	Izel Nakri <contact@izelnakri.com>
Date	2026-03-14 23:51:02 +0100
Message	dev: now we lint for docs & doctests
Files	43
Lines	10871
Repo Size	1.8 MB
FS Size	708.9 MB
FS Owner	runner
FS Group	runner
Generated	2026-03-14 22:54:27 UTC

Table of Contents

File Tree

```
├── .claude
│   ├── commands
│   │   └── doctest.md
│   └── settings.local.json
├── .dockerignore
├── .github
│   └── workflows
│       ├── ci.yml
│       ├── cleanup.yml
│       └── release.yml
├── .gitignore
├── CHANGELOG.md
├── CLAUDE.md
├── Cargo.toml
├── Dockerfile
├── Makefile
├── README.md
├── benches
│   └── pipeline.rs
├── build.rs
├── cliff.toml
├── demo
│   └── demo.tape
├── flake.nix
├── rust-toolchain.toml
├── scripts
│   └── check_benchmarks.py
├── src
│   ├── cli.rs
│   ├── defaults.rs
│   ├── filter.rs
│   ├── git.rs
│   ├── github.rs
│   ├── highlight.rs
│   ├── lib.rs
│   ├── main.rs
│   ├── pdf
│   │   ├── code.rs
│   │   ├── cover.rs
│   │   ├── diff.rs
│   │   ├── fonts.rs
│   │   ├── layout.rs
│   │   ├── mod.rs
│   │   ├── toc.rs
│   │   ├── tree.rs
│   │   ├── user_activity.rs
│   │   ├── user_cover.rs
│   │   └── user_repos.rs
│   ├── preview.rs
│   ├── types.rs
│   └── user_report.rs
└── tests
    └── integration.rs
```

.claude/commands/doctest.md 8 LOC · 636 B · 2026-03-14

```
# doctest

When writing or modifying public functions, methods, or types in this codebase:

1. Every `pub fn`, `pub async fn`, and `pub` method must have a `///` doc comment describing what it does.
2. Every such doc comment must include a `# Examples` section with a working ```` ```rust ```` doctest that compile
3. When editing an existing public item that already has a doctest, keep it — update it only if the signature or be
4. Doctests must be self-contained: import everything they need, construct any required inputs inline, and assert a
```

.claude/settings.local.json

27 LOC · 630 B · 2026-03-02

```
{
  "permissions": {
    "allow": [
      "WebFetch(domain:docs.rs)",
      "WebSearch",
      "WebFetch(domain:github.com)",
      "WebFetch(domain:crates.io)",
      "Bash(cargo search:*)",
      "Bash(cargo test:*)",
      "Bash(cargo clippy:*)",
      "Bash(cargo fmt:*)",
      "Bash(nix flake check:*)",
      "Bash(git status:*)",
      "Bash(git diff:*)",
      "Bash(git log:*)",
      "Bash(git add:*)",
      "Bash(gh run list:*)",
      "Bash(echo:*)",
      "Bash(cargo doc:*)",
      "Bash(wc:*)",
      "Bash(grep:*)",
      "Bash(cargo tree:*)",
      "Bash(make check:*)",
      "Bash(cargo check:*)"
    ]
  }
}
```

.dockerignore

5 LOC · 45 B · 2026-03-02

```
target/
.git/
.github/
bench-baseline/
*.pdf
```

.github/workflows/ci.yml 251 LOC · 9.8 KB · 2026-03-14

```yaml
name: CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  # ── Lint (fast) ──────────────────────────────────────────────────────
  # Surfaces formatting and clippy failures in ~1 min, independently of tests.
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
      - uses: Swatinem/rust-cache@v2
      - run: cargo fmt -- --check
      - run: cargo clippy --all-targets -- -D warnings

  # ── Coverage ─────────────────────────────────────────────────────────
  # Runs all tests (via cargo llvm-cov nextest) and fails if line coverage drops
  # below 85%. Generates an HTML report uploaded as an artifact for the pages job.
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
        with:
          components: llvm-tools-preview
      - uses: Swatinem/rust-cache@v2
      - uses: taiki-e/install-action@v2
        with:
          tool: cargo-llvm-cov,nextest
      - run: cargo llvm-cov nextest --lcov --output-path lcov.info --fail-under-lines 85
      - run: cargo llvm-cov report --html
      - uses: codecov/codecov-action@v5
        with:
          files: lcov.info
          fail_ci_if_error: false
      - uses: actions/upload-artifact@v4
        with:
          name: coverage-report
          path: target/llvm-cov/html/
          retention-days: 1
      - name: Upload coverage reports to Codecov
        uses: codecov/codecov-action@v5
        with:
          token: ${{ secrets.CODECOV_TOKEN }}

  # ── Binary ───────────────────────────────────────────────────────────
  # Compiles a musl release binary in parallel with lint/test. Uses thin LTO
  # (acceptable for nightly; fat LTO is reserved for shipped release binaries).
  # The package and pages jobs download this artifact instead of recompiling.
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
      - name: Install musl toolchain
        run: sudo apt-get install -y musl-tools
      - run: rustup target add x86_64-unknown-linux-musl
      - uses: Swatinem/rust-cache@v2
        with:
          key: musl
      - name: Build
        env:
          CARGO_PROFILE_RELEASE_LTO: thin
          CARGO_PROFILE_RELEASE_CODEGEN_UNITS: "16"
        run: cargo build --release --target x86_64-unknown-linux-musl
      - run: tar czf gitprint-linux-musl.tar.gz -C target/x86_64-unknown-linux-musl/release gitprint
      - uses: actions/upload-artifact@v4
        with:
          name: gitprint-linux-musl
          path: gitprint-linux-musl.tar.gz
          retention-days: 1

```

```yaml
  # ── Docker image ──────────────────────────────────────────────────────────
  # Downloads the pre-built musl binary from the build job and copies it into
  # Alpine via --target prebuilt — no Rust compilation inside Docker.
  #
  # Push policy:
  #   - main branch  → ghcr.io/.../gitprint:nightly + sha-<hash>
  #   - other branches / same-repo PRs → ghcr.io/.../gitprint:<branch> or pr-<n>
  #   - fork PRs → build only, no push (fork tokens lack GHCR write access)
  package:
    needs: [lint, test, build]
    runs-on: ubuntu-latest
    permissions:
      packages: write
    steps:
      - uses: actions/checkout@v4

      - uses: docker/setup-buildx-action@v3

      # Login is skipped for fork PRs — their GITHUB_TOKEN cannot write to the
      # upstream registry. Same-repo PRs and all push events have write access.
      - uses: docker/login-action@v3
        if: >-
          github.event_name == 'push' ||
          github.event.pull_request.head.repo.full_name == github.repository
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      # Tags:
      #   push to main  → nightly + sha-<hash>
      #   push to branch → <branch-name> + sha-<hash>
      #   same-repo PR   → pr-<number> + sha-<hash>
      - uses: docker/metadata-action@v5
        id: meta
        with:
          images: ghcr.io/${{ github.repository }}
          tags: |
            type=raw,value=nightly,enable=${{ github.ref == 'refs/heads/main' }}
            type=ref,event=branch
            type=ref,event=pr
            type=sha,prefix=sha-,format=short

      - uses: actions/download-artifact@v4
        with:
          name: gitprint-linux-musl
          path: dist
      - run: tar xzf dist/gitprint-linux-musl.tar.gz -C dist

      - uses: docker/build-push-action@v6
        with:
          context: dist
          file: Dockerfile
          target: prebuilt
          push: >-
            ${{
              github.event_name == 'push' ||
              github.event.pull_request.head.repo.full_name == github.repository
            }}
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}

  # ── Pages (main only) ─────────────────────────────────────────────────────
  # Reuses the pre-built musl binary to generate the source PDF (skips
  # cargo build --release), then builds rustdoc and deploys both to GitHub Pages.
  # Runs only on main; concurrency guard cancels stale in-flight deployments.
  pages:
    if: github.ref == 'refs/heads/main'
    needs: [lint, test, build]
    runs-on: ubuntu-latest
    concurrency:
      group: pages
      cancel-in-progress: true
    permissions:
      contents: read
      pages: write
      id-token: write
    environment:
```

```yaml
      name: github-pages
      url: ${{ steps.deploy.outputs.page_url }}
    steps:
      - uses: actions/checkout@v4
        with:
          # Full history required so git log reports accurate per-file dates.
          fetch-depth: 0

      - uses: dtolnay/rust-toolchain@stable
      - uses: Swatinem/rust-cache@v2

      # Reuse the already-compiled musl binary — no cargo build --release needed.
      - uses: actions/download-artifact@v4
        with:
          name: gitprint-linux-musl
          path: dist
      - run: tar xzf dist/gitprint-linux-musl.tar.gz -C dist && chmod +x dist/gitprint

      - uses: actions/download-artifact@v4
        with:
          name: coverage-report
          path: coverage-report

      - name: Generate PDF
        run: ./dist/gitprint . --output gitprint.pdf

      - name: Generate rustdoc
        run: cargo doc --no-deps

      - name: Assemble site
        env:
          REPO: ${{ github.repository }}
        run: |
          mkdir -p _site/docs _site/coverage
          cp gitprint.pdf _site/gitprint.pdf
          cp -r target/doc/. _site/docs/
          cp -r coverage-report/. _site/coverage/
          touch _site/.nojekyll
          cat > _site/index.html << HTMLEOF
          <!DOCTYPE html>
          <html lang="en">
          <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width, initial-scale=1.0">
            <title>gitprint — source PDF</title>
            <style>
              * { margin: 0; padding: 0; box-sizing: border-box; }
              html, body { height: 100%; font-family: system-ui, -apple-system, sans-serif; background: #0d1117;
              body { display: flex; flex-direction: column; }
              header { flex-shrink: 0; display: flex; align-items: center; gap: 16px; padding: 10px 20px; backgro
              header h1 { font-size: 15px; font-weight: 600; letter-spacing: -.01em; }
              header a { font-size: 13px; color: #8b949e; text-decoration: none; }
              header a:hover { color: #e6edf3; }
              embed { flex: 1; width: 100%; border: none; }
            </style>
          </head>
          <body>
            <header>
              <h1>gitprint</h1>
              <a href="https://github.com/${REPO}">source</a>
              <a href="gitprint.pdf">download PDF</a>
              <a href="docs/gitprint/index.html">API docs</a>
              <a href="coverage/index.html">coverage</a>
            </header>
            <embed src="gitprint.pdf" type="application/pdf">
          </body>
          </html>
          HTMLEOF

      - uses: actions/configure-pages@v4
      - uses: actions/upload-pages-artifact@v3
        with:
          path: _site
      - id: deploy
        uses: actions/deploy-pages@v4

  # ── Benchmarks (main only) ──────────────────────────────────────────────
  # Run all benchmarks and record results as "current". No regression
```

```yaml
  # comparison here — GitHub-hosted runners have too much performance
  # variance between runs to produce a reliable baseline. The authoritative
  # check runs locally via `make bench-check` / `make release`, where
  # hardware is consistent. Skipped on PRs since results would be discarded.
  benchmark:
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: dtolnay/rust-toolchain@stable
      - uses: Swatinem/rust-cache@v2
      - name: Run benchmarks
        run: cargo bench --bench pipeline -- --save-baseline current
      - uses: actions/upload-artifact@v4
        with:
          name: benchmark-report
          path: target/criterion/
          retention-days: 90
```

.github/workflows/cleanup.yml

31 LOC · 1.1 KB · 2026-03-04

```yaml
name: Cleanup GHCR

# Runs weekly and on demand. Deletes old container versions that are not
# protected by a semver tag, `latest`, or `nightly`. This covers ephemeral
# branch and PR images that accumulate over time.
#
# Protected tag regex: latest | nightly | X.Y.Z | X.Y
# Everything else (sha-*, branch names, pr-*) is subject to deletion once
# the number of unprotected versions exceeds min-versions-to-keep.

on:
  schedule:
    - cron: '0 3 * * 0'  # every Sunday at 03:00 UTC
  workflow_dispatch:

jobs:
  cleanup:
    runs-on: ubuntu-latest
    permissions:
      packages: write
    steps:
      - uses: actions/delete-package-versions@v5
        with:
          package-name: gitprint
          package-type: container
          token: ${{ secrets.GITHUB_TOKEN }}
          # Keep at least the 25 most-recent unprotected versions so in-flight
          # branch images stay available for a reasonable window before deletion.
          min-versions-to-keep: 25
          # Never delete semver releases, latest, or the nightly image.
          ignore-versions: '^(latest|nightly|\d+\.\d+(\.\d+)?)$'
```

.github/workflows/release.yml

```yaml
name: Release Binaries

# Builds native binaries for every major platform and publishes them on
# versioned tag pushes (e.g. v1.2.3). Nightly Docker images are published
# on every main-branch commit by ci.yml — no duplication needed here.

on:
  push:
    tags: ['v*']

env:
  CARGO_TERM_COLOR: always

jobs:
  # ── Build ──────────────────────────────────────────────────────────────
  build:
    name: Build · ${{ matrix.target }}
    runs-on: ${{ matrix.os }}
    strategy:
      fail-fast: false
      matrix:
        include:
          # Linux glibc — preferred by binstall on mainstream distros; uses glibc's fast allocator
          - target: x86_64-unknown-linux-gnu
            os: ubuntu-latest
            cross: false

          # Linux musl — statically linked, runs on Alpine / musl-based distros
          - target: x86_64-unknown-linux-musl
            os: ubuntu-latest
            cross: false

          - target: aarch64-unknown-linux-musl
            os: ubuntu-latest
            cross: true   # cross-compiled via the `cross` Docker toolchain

          # macOS ARM (macos-14+ runner is ARM64)
          - target: aarch64-apple-darwin
            os: macos-latest
            cross: false

          # Windows
          - target: x86_64-pc-windows-msvc
            os: windows-latest
            cross: false

    steps:
      - uses: actions/checkout@v4

      # Toolchain version comes from rust-toolchain.toml; targets are per-matrix.
      - uses: dtolnay/rust-toolchain@stable
        with:
          targets: ${{ matrix.target }}

      # Belt-and-suspenders: ensure the target stdlib is installed even on a cold
      # tool-cache (e.g. after a version bump in rust-toolchain.toml). cross builds
      # use their own Docker image so only native builds need this.
      - name: Ensure Rust target stdlib is installed
        if: "!matrix.cross"
        run: rustup target add ${{ matrix.target }}

      - uses: Swatinem/rust-cache@v2
        with:
          key: release-${{ matrix.target }}

      # musl-tools provides musl-gcc required for the x86_64-musl target.
      - name: Install musl toolchain
        if: contains(matrix.target, 'musl') && !matrix.cross
        run: sudo apt-get install -y musl-tools

      # taiki-e/install-action downloads pre-built cross binaries — much faster than
      # compiling from source with `cargo install cross`.
      - name: Install cross
        if: matrix.cross
        uses: taiki-e/install-action@v2
        with:
          tool: cross
```

```yaml

      - name: Build
        shell: bash
        env:
          # Explicitly reinforce all profile.release optimizations from Cargo.toml so
          # they can't be silently overridden by toolchain defaults or stale cache.
          CARGO_PROFILE_RELEASE_LTO: fat
          CARGO_PROFILE_RELEASE_CODEGEN_UNITS: "1"
          CARGO_PROFILE_RELEASE_OPT_LEVEL: "3"
          CARGO_PROFILE_RELEASE_PANIC: abort
          CARGO_PROFILE_RELEASE_STRIP: symbols
        run: |
          if [[ "${{ matrix.cross }}" == "true" ]]; then
            cross build --release --target ${{ matrix.target }}
          else
            cargo build --release --target ${{ matrix.target }}
          fi

      - name: Package (Unix)
        if: runner.os != 'Windows'
        run: >
          tar czf gitprint-${{ matrix.target }}.tar.gz
          -C target/${{ matrix.target }}/release gitprint

      - name: Package (Windows)
        if: runner.os == 'Windows'
        shell: pwsh
        run: >
          Compress-Archive
          -Path   target/${{ matrix.target }}/release/gitprint.exe
          -DestinationPath gitprint-${{ matrix.target }}.zip

      - uses: actions/upload-artifact@v4
        with:
          name: gitprint-${{ matrix.target }}
          # Each job uploads exactly one archive; glob picks whichever extension was created.
          path: gitprint-${{ matrix.target }}.*

  # ── Docker image ──────────────────────────────────────────────────────────
  # Publishes ghcr.io/izelnakri/gitprint:<version>, <major>.<minor>, and latest.
  package:
    name: Publish Docker image
    needs: [build]
    runs-on: ubuntu-latest
    permissions:
      packages: write
    steps:
      - uses: actions/checkout@v4

      - uses: docker/setup-buildx-action@v3

      - uses: docker/login-action@v3
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      # Derives: v0.3.4  →  tags 0.3.4, 0.3, and latest.
      - uses: docker/metadata-action@v5
        id: meta
        with:
          images: ghcr.io/${{ github.repository }}
          tags: |
            type=semver,pattern={{version}}
            type=semver,pattern={{major}}.{{minor}}
            type=raw,value=latest

      # Download the already-compiled musl binary from the build job — no recompilation needed.
      - uses: actions/download-artifact@v4
        with:
          name: gitprint-x86_64-unknown-linux-musl
          path: dist
      - run: tar xzf dist/gitprint-x86_64-unknown-linux-musl.tar.gz -C dist

      # Target the prebuilt stage: copies the binary into Alpine, skips all compile stages.
      - uses: docker/build-push-action@v6
        with:
          context: dist
```

```yaml
          file: Dockerfile
          target: prebuilt
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}

  # ── Publish ──────────────────────────────────────────────────────────────
  publish:
    name: Publish release
    needs: [build]
    runs-on: ubuntu-latest
    permissions:
      contents: write   # required to create/delete GitHub releases and tags

    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 1

      # Download all platform archives into a single directory.
      - uses: actions/download-artifact@v4
        with:
          merge-multiple: true
          path: artifacts

      - name: Generate SHA-256 checksums
        run: cd artifacts && sha256sum * > checksums.txt

      # make release creates the GitHub Release entry locally (with CHANGELOG notes)
      # before pushing the tag, so CI only needs to upload the built artifacts.
      # The create fallback handles the rare case where CI wins the race.
      - name: Upload artifacts to versioned release
        env:
          GH_TOKEN: ${{ github.token }}
          GH_REPO: ${{ github.repository }}
        run: |
          gh release create "${{ github.ref_name }}" \
            --title "${{ github.ref_name }}" \
            --generate-notes 2>/dev/null || true
          gh release upload "${{ github.ref_name }}" --clobber artifacts/*
```

.gitignore

5 LOC · 43 B · 2026-02-19

```
/target
/result
/bench-baseline
*.pdf
TODO
```

CHANGELOG.md 169 LOC · 11.5 KB · 2026-03-04

```
# Changelog

## [0.4.0] - 2026-03-04
[`v0.3.23...v0.4.0`](https://github.com/izelnakri/gitprint/compare/v0.3.23...v0.4.0)

### Features
- --preview mode!! — 2026-03-04 by [@izelnakri](https://github.com/izelnakri) ([`82eb371`](https://github.com/ize

## [0.3.23] - 2026-03-04
[`v0.3.21...v0.3.23`](https://github.com/izelnakri/gitprint/compare/v0.3.21...v0.3.23)

### Bug Fixes
- V0.3.22 already exists — 2026-03-04 by [@izelnakri](https://github.com/izelnakri) ([`0184026`](https://github.c

## [0.3.21] - 2026-03-04
[`v0.3.20...v0.3.21`](https://github.com/izelnakri/gitprint/compare/v0.3.20...v0.3.21)

### Performance
- Improve CI by caching musl for cargo release — 2026-03-04 by [@izelnakri](https://github.com/izelnakri) ([`b4a9

## [0.3.20] - 2026-03-03
[`v0.3.19...v0.3.20`](https://github.com/izelnakri/gitprint/compare/v0.3.19...v0.3.20)

### Performance
- Parallelise CI, add nextest, skip nightly binary builds on main push — 2026-03-03 by [@izelnakri](https://githu

## [0.3.18] - 2026-03-03
[`v0.3.17...v0.3.18`](https://github.com/izelnakri/gitprint/compare/v0.3.17...v0.3.18)

### Performance
- Override fat LTO in Docker builds to cut link time from 3+ min to ~20 s — 2026-03-03 by [@izelnakri](https://gi

## [0.3.17] - 2026-03-03
[`v0.3.16...v0.3.17`](https://github.com/izelnakri/gitprint/compare/v0.3.16...v0.3.17)

### Bug Fixes
- Use pre-built lukemathwalker/cargo-chef image instead of cargo install — 2026-03-03 by [@izelnakri](https://git

### Performance
- Use cargo-chef + pinned Rust in Dockerfile for reliable layer caching — 2026-03-03 by [@izelnakri](https://gith

## [0.3.16] - 2026-03-03
[`v0.3.15...v0.3.16`](https://github.com/izelnakri/gitprint/compare/v0.3.15...v0.3.16)

### Bug Fixes
- Add stub benches/pipeline.rs to Dockerfile deps layer — 2026-03-03 by [@izelnakri](https://github.com/izelnakri

### Performance
- Replace build-musl + pre-built binary with multi-stage Dockerfile — 2026-03-03 by [@izelnakri](https://github.c

## [0.3.15] - 2026-03-03
[`v0.3.14...v0.3.15`](https://github.com/izelnakri/gitprint/compare/v0.3.14...v0.3.15)

### Bug Fixes
- Drop rust:alpine container for build-musl, use musl-tools instead — 2026-03-03 by [@izelnakri](https://github.c

## [0.3.14] - 2026-03-03
[`v0.3.13...v0.3.14`](https://github.com/izelnakri/gitprint/compare/v0.3.13...v0.3.14)

### Bug Fixes
- Download musl binary before Docker build in release package job — 2026-03-03 by [@izelnakri](https://github.com

## [0.3.13] - 2026-03-03
[`v0.3.12...v0.3.13`](https://github.com/izelnakri/gitprint/compare/v0.3.12...v0.3.13)

### Bug Fixes
- Remove benchmark regression comparison from CI — 2026-03-02 by [@izelnakri](https://github.com/izelnakri) ([`d0
- Remove benchmark regression check from release.yml — 2026-03-02 by [@izelnakri](https://github.com/izelnakri) (

## [0.3.12] - 2026-03-02
[`v0.3.11...v0.3.12`](https://github.com/izelnakri/gitprint/compare/v0.3.11...v0.3.12)

### Bug Fixes
- Mark benchmark job continue-on-error in CI — 2026-03-02 by [@izelnakri](https://github.com/izelnakri) ([`8ad6a6

## [0.3.11] - 2026-03-02
[`v0.3.10...v0.3.11`](https://github.com/izelnakri/gitprint/compare/v0.3.10...v0.3.11)
```

```
78
79  ### Bug Fixes
80  - Use actions/cache with explicit paths for musl build caching — 2026-03-02 by [@izelnakri](https://github.com/iz
81  - Fetch tags from remote before --list-tags on shallow clones — 2026-03-02 by [@izelnakri](https://github.com/ize
82
83  ### Performance
84  - Improve "build-musl" CI workflow time — 2026-03-02 by [@izelnakri](https://github.com/izelnakri) ([`b5f5dd8`](h
85
86  ## [0.3.9] - 2026-03-02
87  [`v0.3.8...v0.3.9`](https://github.com/izelnakri/gitprint/compare/v0.3.8...v0.3.9)
88
89  ### Performance
90  - Run benchmarks in background during changelog preview in make release — 2026-03-02 by [@izelnakri](https://gith
91
92  ## [0.3.8] - 2026-03-02
93  [`v0.3.7...v0.3.8`](https://github.com/izelnakri/gitprint/compare/v0.3.7...v0.3.8)
94
95  ### Bug Fixes
96  - Wire musl-gcc as linker and C compiler for musl CI build — 2026-03-02 by [@izelnakri](https://github.com/izelna
97  - Build musl binary natively in rust:alpine container — 2026-03-02 by [@izelnakri](https://github.com/izelnakri)
98
99  ### Features
100 - --last-repos, --list-tags, date aliases, offline mock tests — 2026-03-02 by [@izelnakri](https://github.com/ize
101 - Show version and binary size on one line in --help footer — 2026-03-02 by [@izelnakri](https://github.com/izeln
102 - Show git branch as version in cargo run; fix musl CI build — 2026-03-02 by [@izelnakri](https://github.com/izel
103
104 ### Performance
105 - Pre-build musl binary in CI; Docker just copies it — 2026-03-02 by [@izelnakri](https://github.com/izelnakri) (
106
107 ## [0.3.7] - 2026-03-01
108 [`v0.3.6...v0.3.7`](https://github.com/izelnakri/gitprint/compare/v0.3.6...v0.3.7)
109
110 ### Features
111 - Now users can do $ gitprint -u izelnakri --commits 100 — 2026-02-20 by [@izelnakri](https://github.com/izelnakr
112 - Neon diff palette, rate-limit errors & user report refactor — 2026-03-01 by [@izelnakri](https://github.com/ize
113
114 ## [0.3.6] - 2026-02-20
115 [`v0.3.5...v0.3.6`](https://github.com/izelnakri/gitprint/compare/v0.3.5...v0.3.6)
116
117 ### Documentation
118 - Add Docker installation guide & nix-aware make release — 2026-02-19 by [@izelnakri](https://github.com/izelnakr
119
120 ### Features
121 - Fix FS Size display & add docker nightly builds — 2026-02-19 by [@izelnakri](https://github.com/izelnakri) ([`0
122 - -u flag now generates user reports! — 2026-02-20 by [@izelnakri](https://github.com/izelnakri) ([`a849ab9`](htt
123
124 ## [0.3.5] - 2026-02-19
125 [`v0.3.4...v0.3.5`](https://github.com/izelnakri/gitprint/compare/v0.3.4...v0.3.5)
126
127 ### Features
128 - Metadata Page "Size" is now "Repo Size" or "FS Size" — 2026-02-19 by [@izelnakri](https://github.com/izelnakri)
129 - ONLY ALLOW RELEASE OF BINARY mean benchmark < $REGRESSION_TRESHOLD — 2026-02-19 by [@izelnakri](https://github.
130
131 ## [0.3.3] - 2026-02-19
132 [`v0.3.2...v0.3.3`](https://github.com/izelnakri/gitprint/compare/v0.3.2...v0.3.3)
133
134 ### Features
135 - Improve release workflow — 2026-02-18 by [@izelnakri](https://github.com/izelnakri) ([`e564a7e`](https://github
136 - Optimized release binaries, interactive release flow, CHANGELOG commit links — 2026-02-18 by [@izelnakri](https
137
138 ## [0.3.2] - 2026-02-18
139 [`v0.3.1...v0.3.2`](https://github.com/izelnakri/gitprint/compare/v0.3.1...v0.3.2)
140
141 ### Features
142 - Add make fix, cargo binstall support, and faster nix run — 2026-02-18 by [@izelnakri](https://github.com/izelna
143 - Accept remote URLs as input (git clone + generate PDF) — 2026-02-18 by [@izelnakri](https://github.com/izelnakr
144 - Improve Metadata first page — 2026-02-18 by [@izelnakri](https://github.com/izelnakri) ([`c7fc8d7`](https://git
145
146 ## [0.3.1] - 2026-02-18
147 [`v0.3.0...v0.3.1`](https://github.com/izelnakri/gitprint/compare/v0.3.0...v0.3.1)
148
149 ### Features
150 - Slim down dependency tree further — 2026-02-18 by [@izelnakri](https://github.com/izelnakri) ([`88ccf92`](https
151
152 ## [0.2.0] - 2026-02-18
153 [`v0.1.2...v0.2.0`](https://github.com/izelnakri/gitprint/compare/v0.1.2...v0.2.0)
154
155 ### Bug Fixes
```

```
- Ensure musl target stdlib is installed on cold cache; add README badges — 2026-02-18 by [@izelnakri](https://gi

### Features
- Add Makefile, release tooling, async save_pdf, and crates.io metadata — 2026-02-18 by [@izelnakri](https://gith

### Refactoring
- Migrate to anyhow, update all deps, parallelise pipeline, add doctests — 2026-02-18 by [@izelnakri](https://git

## [0.1.2] - 2026-02-18

### Features
- Now gitprint shows more metadata of files — 2026-02-18 by [@izelnakri](https://github.com/izelnakri) ([`514f0cb


```

CLAUDE.md 47 LOC · 2.0 KB · 2026-03-04

```
# gitprint

Rust CLI that converts git repositories into syntax-highlighted, printer-friendly PDFs.

## Build & Test

- `make check` — fmt check + clippy + tests (run before every commit)
- `make build` — build the project
- `make test` — run all tests (unit + integration)
- `make fmt` — format source code
- `make doc` — build and open API docs
- `make release [LEVEL=patch|minor|major]` — generate CHANGELOG and publish
- `nix flake check` — run all CI checks (build, clippy, fmt, tests)
- `nix develop` — enter development shell (includes git-cliff, cargo-release)

## Architecture

Single-binary CLI. Pipeline: git → filter → highlight → PDF.

Modules:
- `cli.rs` — Clap argument parsing
- `types.rs` — Shared data types (Config, RepoMetadata, PaperSize, etc.)
- `git.rs` — Git operations via `git` CLI subprocess
- `filter.rs` — Glob-based file filtering + binary/minified detection
- `defaults.rs` — Default exclude glob patterns
- `highlight.rs` — Syntax highlighting via syntect
- `pdf/` — PDF generation via printpdf
  - `mod.rs` — Document creation and writing
  - `fonts.rs` — Embedded JetBrains Mono font loading
  - `cover.rs` — Cover page rendering
  - `toc.rs` — Table of contents rendering
  - `tree.rs` — Directory tree visualization
  - `code.rs` — Highlighted source code rendering
- `lib.rs` — Main pipeline orchestration
- `main.rs` — CLI entry point

## Conventions

- Edition 2024
- Error handling: anyhow for ergonomic error propagation throughout. Always try to use `?` instead of `.unwrap()`,
it is possible.
- Have flat `if` checks if performance characteristics are same between 2 possible control flows.
- Tests: inline `#[cfg(test)] mod tests` for unit tests, `tests/` directory for integration tests. Make sure each
is well covered with tests.
- Integration tests use `tempfile` crate to create temporary git repos
- No unsafe code, if not needed.
- Code has to be always readable. Priorities are: Performance > Scalability > Readability > Maintainance
```

Cargo.toml

52 LOC · 1.8 KB · 2026-03-04

```
[package]
name = "gitprint"
version = "0.4.0"
edition = "2024"
description = "Convert git repositories into beautifully formatted, printer-friendly PDFs"
license = "MIT"
repository = "https://github.com/izelnakri/gitprint"
homepage = "https://github.com/izelnakri/gitprint"
readme = "README.md"
keywords = ["pdf", "git", "syntax-highlighting", "printing", "code"]
categories = ["command-line-utilities", "development-tools"]

[dependencies]
anyhow = "1"
clap = { version = "4", features = ["derive"] }
globset = "0.4"
printpdf = "0.9"
reqwest = { version = "0.12", default-features = false, features = ["json", "rustls-tls"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tokio = { version = "1", features = ["rt-multi-thread", "macros", "fs", "process"] }
syntect = { version = "5", default-features = false, features = ["default-syntaxes", "default-themes", "regex-fanc

[dev-dependencies]
criterion = { version = "0.8", features = ["html_reports"] }
httpmock = "0.7"
tempfile = "3"
tokio = { version = "1", features = ["rt-multi-thread", "macros", "fs", "process"] }

[[bench]]
name = "pipeline"
harness = false

[package.metadata.release]
# Run git-cliff inside cargo-release's own transaction so CHANGELOG.md is
# written after the dirty check passes and included in the release commit.
pre-release-hook = ["git-cliff", "--tag", "v{{version}}", "-o", "CHANGELOG.md"]
# make release already runs `make check` (fmt + clippy + tests) immediately
# before invoking cargo-release, so skipping the duplicate test run here
# saves ~1-2 min without reducing safety.
verify = false

[package.metadata.binstall]
pkg-url = "{ repo }/releases/download/v{ version }/{ name }-{ target }{ archive-suffix }"
bin-dir = "{ bin }{ binary-ext }"

[profile.release]
opt-level = 3
lto = "fat"
codegen-units = 1
strip = true
panic = "abort"
```

Dockerfile

41 LOC · 1.7 KB · 2026-03-04

```
# cargo-chef is pre-installed in this image — no source compilation needed.
# 0.1.75-rust-alpine3.23 is an immutable tag (pinned cargo-chef + Alpine version)
# so the chef and deps layers are never invalidated by a base image update.
# Bump the tag deliberately when upgrading Rust or cargo-chef.
FROM lukemathwalker/cargo-chef:0.1.75-rust-alpine3.23 AS chef
WORKDIR /app

# Planner: fast — inspects Cargo.toml/Cargo.lock and emits recipe.json.
FROM chef AS planner
COPY . .
RUN cargo chef prepare --recipe-path recipe.json

# Builder stage 1 — dependencies only.
# Cached until recipe.json changes (i.e. Cargo.lock / Cargo.toml change).
# LTO and codegen-units are overridden here: fat LTO + CU=1 cannot be
# layer-cached and would add 3+ min of link time on every push.
FROM chef AS builder
COPY --from=planner /app/recipe.json recipe.json
RUN CARGO_PROFILE_RELEASE_LTO=thin \
    CARGO_PROFILE_RELEASE_CODEGEN_UNITS=16 \
    cargo chef cook --release --recipe-path recipe.json

# Builder stage 2 — gitprint source only (~20-30 s on every push).
COPY . .
RUN CARGO_PROFILE_RELEASE_LTO=thin \
    CARGO_PROFILE_RELEASE_CODEGEN_UNITS=16 \
    cargo build --release

# Pre-built stage: copies an already-compiled musl binary from the build context.
# Used by both ci.yml and release.yml (--target prebuilt) to skip all compilation.
# Local full build: `docker build .` uses the default stage below instead.
FROM alpine:latest AS prebuilt
RUN apk add --no-cache git
COPY gitprint /usr/local/bin/gitprint
ENTRYPOINT ["gitprint"]

# Default stage: compiled from source via the builder above (local docker build).
FROM alpine:latest
RUN apk add --no-cache git
COPY --from=builder /app/target/release/gitprint /usr/local/bin/gitprint
ENTRYPOINT ["gitprint"]
```

Makefile 100 LOC · 4.4 KB · 2026-03-14

```
.DEFAULT_GOAL := help

LEVEL ?= patch
REGRESSION_THRESHOLD ?= 20

.PHONY: help fix check fmt test build doc bench-baseline bench-check release coverage

help:
	@echo "Usage: make <target> [LEVEL=patch|minor|major] [REGRESSION_THRESHOLD=20]"
	@echo ""
	@echo "  fix             Auto-fix formatting and clippy lints"
	@echo "  check           Fmt check + clippy + tests (validation)"
	@echo "  fmt             Format source code"
	@echo "  test            Run all tests"
	@echo "  build           Build the project"
	@echo "  doc             Build and open API documentation"
	@echo "  bench-baseline  Establish (or reset) the local benchmark baseline"
	@echo "  bench-check     Run benchmarks and report regressions vs baseline"
	@echo "  coverage        Run tests with coverage and fail if line coverage < 85%"
	@echo "  release         Preview changelog, confirm, then generate CHANGELOG and publish (LEVEL=patch)"

fix:
	cargo fmt
	cargo clippy --fix --allow-dirty --allow-staged --all-targets

check:
	cargo fmt -- --check
	cargo clippy --all-targets -- -D warnings
	@cargo nextest --version >/dev/null 2>&1 && cargo nextest run || cargo test

fmt:
	cargo fmt

test:
	@cargo nextest --version >/dev/null 2>&1 && cargo nextest run || cargo test

build:
	cargo build

doc:
	cargo doc --no-deps --open

# Run all benchmarks and save results as the new "main" baseline.
# Run this once after cloning, or whenever you want to reset the reference point.
bench-baseline:
	@echo "=== Establishing benchmark baseline ==="
	cargo bench --bench pipeline -- --save-baseline current
	REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py --save
	@echo "Done. Future 'make bench-check' and 'make release' will compare against this baseline."

# Run benchmarks as "current" and compare against the stored baseline.
# Exits non-zero if any benchmark regressed more than REGRESSION_THRESHOLD percent.
bench-check:
	@echo "=== Benchmark regression check (threshold: $(REGRESSION_THRESHOLD)%) ==="
	cargo bench --bench pipeline -- --save-baseline current
	REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py

coverage:
	@cargo nextest --version >/dev/null 2>&1 && cargo llvm-cov nextest --html --open --fail-under-lines 85 || cargo

# Benchmarks run in the background while the developer reads the changelog
# preview — reading time is free CPU time. Results are checked after the
# developer confirms, before cargo-release runs. Aborts kill the background
# process cleanly. `fix` is intentionally excluded: auto-modifying code at
# release time is unsafe; run `make fix` manually if check fails.
# If not already inside a nix shell, re-enters via `nix develop` so git-cliff
# and cargo-release are available. `exec` replaces the shell process, preventing
# any subsequent commands from running outside nix.
release:
	@if [ -z "$$IN_NIX_SHELL" ]; then \
	  echo "==> Entering nix develop (git-cliff, cargo-release)..."; \
	  exec nix develop --command $(MAKE) release LEVEL=$(LEVEL) REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD); \
	fi; \
	$(MAKE) check; \
	cargo bench --bench pipeline -- --save-baseline current > /tmp/gitprint-bench.log 2>&1 & BENCH_PID=$$!; \
	printf "\n=== Release Preview (level: $(LEVEL)) ===\n\n"; \
	git-cliff --bump --unreleased 2>/dev/null || true; \
```

```
	printf "\nProceed with $(LEVEL) release? [y/N] " > /dev/tty; \
	read confirm < /dev/tty; \
	case "$$confirm" in \
	  [yY]*) \
	    printf "\n=== Benchmark results ===\n"; \
	    tail -f /tmp/gitprint-bench.log & TAIL_PID=$$!; \
	    wait $$BENCH_PID; BENCH_EXIT=$$?; \
	    kill $$TAIL_PID 2>/dev/null; wait $$TAIL_PID 2>/dev/null; \
	    [ $$BENCH_EXIT -eq 0 ] || { printf "\nBenchmark run failed.\n"; rm -f /tmp/gitprint-bench.log; exit 1; }; \
	    REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py || { rm -f /tmp/gitprint-be
	    cargo release $(LEVEL) --execute; \
	    TAG=$$(git describe --tags --abbrev=0); \
	    awk '/^## \[/{if(found) exit; found=1} found' CHANGELOG.md > /tmp/release-notes.md; \
	    gh release create "$$TAG" --title "$$TAG" --notes-file /tmp/release-notes.md; \
	    rm -f /tmp/release-notes.md /tmp/gitprint-bench.log; \
	    REGRESSION_THRESHOLD=$(REGRESSION_THRESHOLD) python3 scripts/check_benchmarks.py --save \
	  ;; \
	  *) printf "Aborted.\n"; kill $$BENCH_PID 2>/dev/null; rm -f /tmp/gitprint-bench.log; exit 1 ;; \
	esac

demo:
	vhs demo/demo.tape

```

README.md 241 LOC · 7.7 KB · 2026-03-14

```
<div align="center">

# gitprint

[![CI](https://github.com/izelnakri/gitprint/actions/workflows/ci.yml/badge.svg)](https://github.com/izelnakri/gi
[![Coverage](https://codecov.io/gh/izelnakri/gitprint/graph/badge.svg)](https://codecov.io/gh/izelnakri/gitprint)
[![Crate](https://img.shields.io/crates/v/gitprint)](https://crates.io/crates/gitprint)
[![Downloads](https://img.shields.io/crates/d/gitprint)](https://crates.io/crates/gitprint)
[![Docs](https://img.shields.io/badge/docs-online-blue)](https://izelnakri.github.io/gitprint/docs/gitprint/)
[![Sponsor](https://img.shields.io/badge/sponsor-%E2%99%A5-pink)](https://github.com/sponsors/izelnakri)

</div>

Convert git repositories into beautifully formatted, printer-friendly PDFs — or preview them directly in the term

![gitprint demo](demo/demo.gif)

## Features

- Syntax-highlighted source code with 100+ languages supported
- Configurable color themes (InspiredGitHub, Solarized, base16, and more)
- Table of contents and directory tree visualization
- Single-file mode — print just one file, no cover page or TOC overhead
- Plain directory support — works on any folder, not just git repos
- Automatic binary and minified file detection and exclusion
- Glob-based include/exclude filtering
- Multiple paper sizes (A4, Letter, Legal) and landscape mode
- Branch and commit selection for printing specific revisions
- Embedded JetBrains Mono font for crisp code rendering
- Async pipeline — metadata, file reads, and highlighting run concurrently
- **Terminal preview mode** — inspect repo or user data in the terminal without generating a PDF
- **GitHub user report mode** — generate a PDF (or preview) of a user's activity, repos, and recent commits

## Installation

### cargo-binstall (recommended — downloads pre-built binary)

```sh
cargo binstall gitprint
```

Pre-built binary

Download the latest release for your platform from the
[releases page](https://github.com/izelnakri/gitprint/releases).

Nix

```sh
nix profile install github:izelnakri/gitprint
```

Build from source

```sh
cargo install --git https://github.com/izelnakri/gitprint
```

With Docker

No install needed — pull the latest nightly image from GitHub Container Registry and mount your repository:

```sh
docker run --rm -w /repo -v "$(pwd):/repo" ghcr.io/izelnakri/gitprint:nightly . -o output.pdf
```

- `--rm` removes the container after use
- `-w /repo` sets the working directory inside the container so `.` resolves correctly
- `-v "$(pwd):/repo"` mounts the current directory; the output PDF is written back to it

Usage

Repository Mode (Default)

```sh
# Generate PDF from current directory (or git repo)
gitprint .
```
```

```

# Print a single file
gitprint src/main.rs

# Print any directory (no git required)
gitprint /path/to/dir

# Print a remote repository
gitprint https://github.com/user/repo

# Output to a specific file
gitprint . -o output.pdf

# Include only Rust and TOML files
gitprint . --include "*.rs" --include "*.toml"

# Exclude test files
gitprint . --exclude "test_*.rs"

# Use a different theme
gitprint . --theme "Solarized (dark)"

# List available themes
gitprint . --list-themes

# Use Letter paper in landscape
gitprint . --paper-size letter --landscape

# Print a specific branch or commit
gitprint . --branch feature-x
gitprint . --commit abc1234

# Minimal output: no TOC, no file tree, no line numbers
gitprint . --no-toc --no-file-tree --no-line-numbers
```

User Report Mode

```sh
# Generate a GitHub user activity report PDF
gitprint --user torvalds

# Limit the activity date range
gitprint --user torvalds --since "last month"
gitprint --user torvalds --since 2024-01-01 --until 2024-12-31

# Show only push events (commits), not issues/PRs/stars
gitprint --user torvalds --activity commits

# Include more repos and commits in the report
gitprint --user torvalds --last-repos 10 --last-commits 10

# Skip commit diffs for a faster, lighter report
gitprint --user torvalds --no-diffs

# Increase GitHub API rate limits with a personal access token
GITHUB_TOKEN=ghp_... gitprint --user torvalds
```

Preview Mode

Preview shows all the same data as the PDF — metadata, directory tree, file list with LOC/sizes, or GitHub user a

```sh
# Preview a repository in the terminal
gitprint . --preview

# Preview a remote repository
gitprint https://github.com/user/repo --preview

# Preview a GitHub user report
gitprint --user torvalds --preview

# Combine with other flags — filters and date ranges all apply
gitprint . --preview --include "*.rs"
gitprint --user torvalds --preview --since "last month" --activity commits
```

CLI Reference

```
Convert git repositories into beautifully formatted PDFs.

MODES

  gitprint <PATH> [OPTIONS]
    Local path, file, or remote URL (https://, git@, ssh://) → PDF

  gitprint --user <USERNAME> [OPTIONS]
    GitHub user activity report → PDF

  gitprint <PATH|--user USERNAME> --preview
    Preview output in the terminal — no PDF generated

Usage: gitprint [OPTIONS] [PATH]

Arguments:
  [PATH]
    Local path, file, or remote URL (https://, git@, ssh://)

Options:
      --preview          Preview output in the terminal instead of generating a PDF
  -o, --output <PATH>    Output PDF file path
  -h, --help             Print help
  -V, --version          Print version

Repository Mode (Default):
      --include <PATTERN>      Glob patterns for files to include (repeatable)
      --exclude <PATTERN>      Glob patterns for files to exclude (repeatable)
      --theme <NAME>           Syntax highlighting theme [default: InspiredGitHub]
      --font-size <SIZE>       Code font size in points [default: 8]
      --no-line-numbers        Disable line numbers
      --no-toc                 Disable table of contents
      --no-file-tree           Disable directory tree visualization
      --branch <NAME>          Use a specific branch
      --commit <HASH>          Use a specific commit
      --paper-size <SIZE>      Paper size [default: a4] [possible values: a4, letter, legal]
      --landscape              Use landscape orientation
      --list-themes            List available syntax themes and exit
      --list-tags              List version tags of the repository and exit

User Report Mode:
  -u, --user <USERNAME>        GitHub username — generate a user activity report
      --last-repos <N>         Most-recently-pushed repos to include [default: 5]
      --last-commits <N>       Recent commits with diffs to render [default: 5]
      --no-diffs               Skip commit diff rendering (faster)
      --since <DATE>           Show events from this date forward
      --until <DATE>           Show events up to and including this date
      --activity <TYPE>        Event types: all (default) or commits
      --events <N>             Max events shown in activity feed [default: 30]
```

Date formats for `--since` / `--until`

Format	Example
ISO date	`2024-01-15` or `2024-01-15T00:00:00Z`
Keywords	`today`, `yesterday`
Named	`last week`, `last month`, `last year`
Relative	`30 days ago`, `2 weeks ago`, `1 month ago`

Development

```sh
nix develop        # Enter dev shell (Rust toolchain, git-cliff, cargo-release)
make check         # Fmt check + clippy + tests (run before every commit)
make build         # Build
make test          # Run tests
make doc           # Build and open API docs
make release       # Bump CHANGELOG + publish (LEVEL=patch|minor|major)
nix flake check    # Full CI suite: build, clippy, fmt, tests
```

Donate

If gitprint saves you time, consider sponsoring development:

[github.com/sponsors/izelnakri](*https://github.com/sponsors/izelnakri*)

GitHub Sponsors has zero platform fees — 100% goes to the developer.

License

MIT

benches/pipeline.rs 91 LOC · 2.4 KB · 2026-02-18

```rust
use std::path::Path;

use std::hint::black_box;

use criterion::{Criterion, criterion_group, criterion_main};

use gitprint::filter::FileFilter;
use gitprint::highlight::Highlighter;
use gitprint::types::HighlightedLine;

const SAMPLE_RUST: &str = r#"
use std::collections::HashMap;

fn main() {
    let mut map = HashMap::new();
    map.insert("key", 42);

    for (k, v) in &map {
        println!("{k}: {v}");
    }

    let result: Vec<_> = (0..100)
        .filter(|n| n % 2 == 0)
        .map(|n| n * n)
        .collect();

    println!("{result:?}");
}
"#;

fn bench_highlight(c: &mut Criterion) {
    let highlighter = Highlighter::new("InspiredGitHub").unwrap();
    let path = Path::new("sample.rs");

    c.bench_function("highlight_rust_file", |b| {
        b.iter(|| {
            let lines: Vec<HighlightedLine> = highlighter
                .highlight_lines(black_box(SAMPLE_RUST), path)
                .collect();
            black_box(lines);
        });
    });

    let large_content = SAMPLE_RUST.repeat(50);
    c.bench_function("highlight_large_file", |b| {
        b.iter(|| {
            let lines: Vec<HighlightedLine> = highlighter
                .highlight_lines(black_box(&large_content), path)
                .collect();
            black_box(lines);
        });
    });
}

fn bench_filter(c: &mut Criterion) {
    let paths: Vec<std::path::PathBuf> = (0..1000)
        .flat_map(|i| {
            vec![
                format!("src/module_{i}/mod.rs").into(),
                format!("src/module_{i}/test.rs").into(),
                format!("docs/page_{i}.md").into(),
                format!("assets/image_{i}.png").into(),
                format!("node_modules/pkg_{i}/index.js").into(),
            ]
        })
        .collect();

    c.bench_function("filter_5000_paths", |b| {
        b.iter(|| {
            let filter = FileFilter::new(&["*.rs".to_string()], &["*test*".to_string()]).unwrap();
            let filtered: Vec<_> = filter.filter_paths(black_box(paths.clone())).collect();
            black_box(filtered);
        });
    });
}

fn bench_highlighter_creation(c: &mut Criterion) {
```

```
    c.bench_function("highlighter_new", |b| {
        b.iter(|| {
            black_box(Highlighter::new("InspiredGitHub").unwrap());
        });
    });
}

criterion_group!(
    benches,
    bench_highlight,
    bench_filter,
    bench_highlighter_creation
);
criterion_main!(benches);
```

build.rs 23 LOC · 806 B · 2026-03-02

```rust
fn main() {
    // Rerun whenever the checked-out branch changes.
    println!("cargo:rerun-if-changed=.git/HEAD");

    // Only embed the branch name in debug builds so `cargo run` shows the
    // current branch while release binaries show the Cargo.toml version.
    if std::env::var("PROFILE").as_deref() != Ok("debug") {
        return;
    }

    let branch = std::process::Command::new("git")
        .args(["rev-parse", "--abbrev-ref", "HEAD"])
        .output()
        .ok()
        .filter(|o| o.status.success())
        .and_then(|o| String::from_utf8(o.stdout).ok())
        .map(|s| s.trim().to_string())
        .filter(|s| !s.is_empty() && s != "HEAD"); // HEAD = detached, skip

    if let Some(branch) = branch {
        println!("cargo:rustc-env=GITPRINT_GIT_BRANCH={branch}");
    }
}
```

cliff.toml 64 LOC · 2.8 KB · 2026-03-02

```
[changelog]
header = "# Changelog\n\n"
body = """
{%- set host     = "https://github.com" -%}
{%- set repo_url = host ~ "/" ~ remote.github.owner ~ "/" ~ remote.github.repo -%}
{% if version -%}
## [{{ version | trim_start_matches(pat="v") }}] - {{ timestamp | date(format="%Y-%m-%d") }}
{% if previous.version %}[`{{ previous.version }}...{{ version }}`]({{ repo_url }}/compare/{{ previous.version }}.
{% endif %}
{% else -%}
## [unreleased]
{% endif -%}
{% for group, commits in commits | group_by(attribute="group") -%}
### {{ group | upper_first }}
{% for commit in commits -%}
- {% if commit.breaking %}[**breaking**] {% endif %}{{ commit.message | upper_first }} — {{ commit.author.timestam
{% endfor %}
{% endfor %}\
"""
footer = ""
trim = true

[git]
conventional_commits = true
filter_unconventional = true
split_commits = false
commit_parsers = [
    { message = "^feat",     group = "Features" },
    { message = "^fix",      group = "Bug Fixes" },
    { message = "^perf",     group = "Performance" },
    { message = "^refactor", group = "Refactoring" },
    { message = "^docs?",    group = "Documentation" },
    { message = "^chore",    skip = true },
    { message = "^ci",       skip = true },
    { message = "^test",     skip = true },
    { message = "^build",    skip = true },
]
filter_commits = true
tag_pattern = "v[0-9].*"
skip_tags = "nightly"
topo_order = false
sort_commits = "oldest"

# ── Hosting configuration ──────────────────────────────────────────
# git-cliff reads `owner` and `repo` from this section and exposes them as
# `remote.github.owner` / `remote.github.repo` in the body template above.
# Both values are auto-detected from `git remote get-url origin` when omitted.
#
# To switch providers:
#   1. Rename the section heading below (e.g. [remote.gitlab])
#   2. Update `host` in the body template to match the provider base URL
#   3. If needed, adjust URL path segments in the template:
#        GitLab    /compare/ → /-/compare/    /commit/ → /-/commit/
#        Bitbucket /compare/ → /branches/compare/  /commit/ → /commits/
#
# Provider section names git-cliff understands:
#   [remote.github]    — github.com
#   [remote.gitlab]    — gitlab.com  (or self-hosted GitLab)
#   [remote.gitea]     — gitea.io    (or self-hosted Gitea)
#   [remote.forgejo]   — forgejo.org (or self-hosted Forgejo)
#   [remote.bitbucket] — bitbucket.org
[remote.github]
owner = "izelnakri"
repo  = "gitprint"
```

demo/demo.tape

56 LOC · 890 B · 2026-03-05

```
Output demo/demo.gif

Set Shell "bash"
Set FontSize 16
Set Width 1200
Set Height 700
Set Theme "Catppuccin Mocha"
Set Padding 20
Set BorderRadius 12
Set WindowBar Colorful
Set Framerate 30
Set TypingSpeed 40ms

Hide
Type "export PATH=$HOME/.cargo/bin:$PATH && cd /home/izelnakri/Github/gitprint && clear"
Enter
Sleep 1s
Show

Type "# Generate a PDF of this repository"
Sleep 500ms
Enter
Sleep 300ms
Type "gitprint . -o demo/gitprint.pdf"
Sleep 300ms
Enter
Sleep 5s

Type "xdg-open demo/gitprint.pdf"
Sleep 300ms
Enter
Sleep 2s

Type "# Generate a GitHub user activity report"
Sleep 500ms
Enter
Sleep 300ms
Type "gitprint --user izelnakri -o demo/izelnakri.pdf"
Sleep 300ms
Enter
Sleep 5s

Type "xdg-open demo/izelnakri.pdf"
Sleep 300ms
Enter
Sleep 5s

Type "# Preview the repo structure in the terminal"
Sleep 500ms
Enter
Sleep 300ms
Type "gitprint --preview ."
Sleep 300ms
Enter
Sleep 5s

```

flake.nix 117 LOC · 3.6 KB · 2026-03-14

```
{
  description = "Convert git repositories into beautifully formatted, printer-friendly PDFs";

  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixpkgs-unstable";
    flake-utils.url = "github:numtide/flake-utils";
    crane.url = "github:ipetkov/crane";
    rust-overlay = {
      url = "github:oxalica/rust-overlay";
      inputs.nixpkgs.follows = "nixpkgs";
    };
  };

  outputs = { self, nixpkgs, flake-utils, crane, rust-overlay }:
    flake-utils.lib.eachDefaultSystem (system:
      let
        overlays = [ (import rust-overlay) ];
        pkgs = import nixpkgs { inherit system overlays; };

        # Pin to the exact version declared in rust-toolchain.toml — same as CI.
        # Dev shell adds rust-src + rust-analyzer on top for IDE support.
        rustToolchain = (pkgs.rust-bin.fromRustupToolchainFile ./rust-toolchain.toml).override {
          extensions = [ "rust-src" "rust-analyzer" "llvm-tools-preview" ];
        };

        craneLib = (crane.mkLib pkgs).overrideToolchain rustToolchain;

        # Source filtering: include Rust/Cargo files + embedded fonts
        unfilteredRoot = ./.;
        src = pkgs.lib.fileset.toSource {
          root = unfilteredRoot;
          fileset = pkgs.lib.fileset.unions [
            (craneLib.fileset.commonCargoSources unfilteredRoot)
            (pkgs.lib.fileset.fileFilter (file: file.hasExt "ttf") unfilteredRoot)
          ];
        };

        commonArgs = {
          inherit src;
          nativeBuildInputs = [ pkgs.pkg-config pkgs.makeWrapper ];
        };

        # Build dependencies only — cached separately from source changes
        cargoArtifacts = craneLib.buildDepsOnly commonArgs;

        # Build-only package: no tests so `nix run` and `nix build` are fast.
        # Tests live in checks.tests below and still run on `nix flake check`.
        gitprint = craneLib.buildPackage (commonArgs // {
          inherit cargoArtifacts;
          doCheck = false;

          postInstall = ''
            wrapProgram $out/bin/gitprint \
              --prefix PATH : ${pkgs.lib.makeBinPath [ pkgs.git ]}
          '';

          meta = with pkgs.lib; {
            description = "Convert git repositories into beautifully formatted, printer-friendly PDFs";
            homepage = "https://github.com/izelnakri/gitprint";
            license = licenses.mit;
            mainProgram = "gitprint";
          };
        });
      in
      {
        packages = {
          default = gitprint;
          gitprint = gitprint;
        };

        apps.default = flake-utils.lib.mkApp {
          drv = gitprint;
        };

        devShells.default = pkgs.mkShell {
          packages = [
            rustToolchain
```

```
            pkgs.git
            pkgs.cargo-watch
            pkgs.cargo-edit
            pkgs.git-cliff
            pkgs.cargo-release
            pkgs.cargo-nextest
            pkgs.cargo-llvm-cov
            pkgs.vhs
          ];

          RUST_SRC_PATH = "${rustToolchain}/lib/rustlib/src/rust/library";
        };

        checks = {
          inherit gitprint;

          tests = craneLib.cargoTest (commonArgs // {
            inherit cargoArtifacts;
            nativeCheckInputs = [ pkgs.git ];
            preCheck = ''
              export HOME=$(mktemp -d)
            '';
          });

          clippy = craneLib.cargoClippy (commonArgs // {
            inherit cargoArtifacts;
            cargoClippyExtraArgs = "--all-targets -- -D warnings";
          });

          fmt = craneLib.cargoFmt {
            inherit src;
          };
        };
      }
    ) // {
      overlays.default = final: prev: {
        gitprint = self.packages.${prev.stdenv.hostPlatform.system}.default;
      };
    };
}
```

rust-toolchain.toml

3 LOC · 88 B · 2026-03-14

```
[toolchain]
channel = "1.88.0"
components = ["rustfmt", "clippy", "llvm-tools-preview"]
```

scripts/check_benchmarks.py 101 LOC · 3.3 KB · 2026-02-19

```
#!/usr/bin/env python3
"""Benchmark regression checker for gitprint.

Modes
-----
Default (no flags)
  Compare target/criterion/**/current/estimates.json against the stored
  baseline in bench-baseline/**/main/estimates.json.  Exits 1 if any
  benchmark regressed more than REGRESSION_THRESHOLD percent (default 20).

--save
  Promote target/criterion/**/current/ results to bench-baseline/**/main/
  without running any comparison.  Call this after a passing check to lock
  in the current results as the new baseline.

The REGRESSION_THRESHOLD environment variable overrides the default (20%).

Usage
-----
  # Check for regressions (fails fast):
  python3 scripts/check_benchmarks.py

  # Save current results as the new baseline (after a passing release):
  python3 scripts/check_benchmarks.py --save
"""
import glob
import json
import os
import shutil
import sys

CRITERION_DIR = "target/criterion"
BASELINE_DIR = "bench-baseline"


def _load_mean(path: str) -> float:
    with open(path) as f:
        return json.load(f)["mean"]["point_estimate"]


def save() -> None:
    """Copy target/criterion/**/current/estimates.json → bench-baseline/**/main/…"""
    saved = 0
    for src in glob.glob(f"{CRITERION_DIR}/**/current/estimates.json", recursive=True):
        rel = os.path.relpath(src, CRITERION_DIR)
        dest = os.path.join(BASELINE_DIR, rel.replace("/current/", "/main/"))
        os.makedirs(os.path.dirname(dest), exist_ok=True)
        shutil.copy(src, dest)
        saved += 1
    print(f"Baseline updated: {saved} result(s) saved to {BASELINE_DIR}/.")


def check(threshold_pct: float = 20.0) -> bool:
    """Return True if all benchmarks are within threshold, False otherwise."""
    if not os.path.isdir(BASELINE_DIR):
        print(f"No baseline found in {BASELINE_DIR}/.")
        print("Run 'make bench-baseline' once to establish one, then re-run.")
        print("Skipping regression check for this run.")
        return True  # don't fail on first-ever run

    threshold = threshold_pct / 100.0
    failures: list[tuple[str, float]] = []
    compared = 0

    for cur_path in glob.glob(f"{CRITERION_DIR}/**/current/estimates.json", recursive=True):
        rel = os.path.relpath(cur_path, CRITERION_DIR)
        name = rel.split(os.sep)[0]
        main_path = os.path.join(BASELINE_DIR, rel.replace("/current/", "/main/"))
        if not os.path.exists(main_path):
            print(f"  (no baseline for {name}, skipping)")
            continue
        cur_mean = _load_mean(cur_path)
        main_mean = _load_mean(main_path)
        change = (cur_mean - main_mean) / main_mean
        arrow = "▲" if change > 0 else "▼"
        print(f"  {name}: {arrow}{abs(change) * 100:.1f}%")
        if change > threshold:
```

```
            failures.append((name, change))
        compared += 1

    if compared == 0:
        print("  (no benchmarks to compare)")
        return True

    if failures:
        print(f"\nRegressions exceeding {threshold_pct:.0f}% threshold:")
        for name, change in failures:
            print(f"  FAIL  {name}: +{change * 100:.1f}% slower than baseline")
        return False

    print("All benchmarks within threshold.")
    return True


if __name__ == "__main__":
    if "--save" in sys.argv:
        save()
    else:
        threshold = float(os.getenv("REGRESSION_THRESHOLD", "20"))
        if not check(threshold):
            sys.exit(1)
```

src/cli.rs

```rust
use std::path::PathBuf;

use clap::Parser;

use crate::types::{ActivityFilter, PaperSize};

/// Parsed command-line arguments for the `gitprint` binary.
#[derive(Parser, Debug)]
#[command(
    name = "gitprint",
    about = "Convert git repositories into beautifully formatted PDFs",
    long_about = "Convert git repositories into beautifully formatted PDFs.\n\
                  \n\
                  MODES\n\
                  \n  \
                  gitprint <PATH> [OPTIONS]\n    \
                    Local path, file, or remote URL (https://, git@, ssh://) → PDF\n\
                  \n  \
                  gitprint --user <USERNAME> [OPTIONS]\n    \
                    GitHub user activity report → PDF\n\
                  \n  \
                  gitprint <PATH|--user USERNAME> --preview\n    \
                    Preview output in the terminal — no PDF generated",
    version,
    arg_required_else_help = true,
    after_help = after_help_text(),
)]
pub struct Args {
    /// Local path, file, or remote URL (https://, git@, ssh://)
    pub path: Option<String>,

    /// Preview output in the terminal instead of generating a PDF
    #[arg(long)]
    pub preview: bool,

    /// Output PDF file path
    #[arg(short, long)]
    pub output: Option<PathBuf>,

    // ── Repository Mode ──────────────────────────────────────────────
    /// Glob patterns for files to include (repeatable)
    #[arg(long, action = clap::ArgAction::Append, help_heading = "Repository Mode (Default)")]
    pub include: Vec<String>,

    /// Glob patterns for files to exclude (repeatable)
    #[arg(long, action = clap::ArgAction::Append, help_heading = "Repository Mode (Default)")]
    pub exclude: Vec<String>,

    /// Syntax highlighting theme
    #[arg(
        long,
        default_value = "InspiredGitHub",
        help_heading = "Repository Mode (Default)"
    )]
    pub theme: String,

    /// Code font size in points
    #[arg(
        long,
        default_value_t = 8.0,
        help_heading = "Repository Mode (Default)"
    )]
    pub font_size: f64,

    /// Disable line numbers
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub no_line_numbers: bool,

    /// Disable table of contents
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub no_toc: bool,

    /// Disable directory tree visualization
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub no_file_tree: bool,

    /// Use a specific branch
```

```rust
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub branch: Option<String>,

    /// Use a specific commit
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub commit: Option<String>,

    /// Paper size
    #[arg(long, value_enum, default_value_t = PaperSize::A4, help_heading = "Repository Mode (Default)")]
    pub paper_size: PaperSize,

    /// Use landscape orientation
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub landscape: bool,

    /// List available syntax themes and exit
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub list_themes: bool,

    /// List version tags of the repository and exit
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub list_tags: bool,

    /// Open the repository in Neovim instead of generating a PDF
    #[arg(long, help_heading = "Repository Mode (Default)")]
    pub nvim: bool,

    // ── User Report Mode ──────────────────────────────────────────────
    /// GitHub username — generate a user activity report instead of printing a repo
    #[arg(short = 'u', long = "user", help_heading = "User Report Mode")]
    pub user: Option<String>,

    /// Number of most-recently-pushed repos to include [default: 5]
    #[arg(long, default_value_t = 5, help_heading = "User Report Mode")]
    pub last_repos: usize,

    /// Number of recent commits with diffs to render [default: 5]
    #[arg(long, default_value_t = 5, help_heading = "User Report Mode")]
    pub last_commits: usize,

    /// Skip commit diff rendering (faster)
    #[arg(long, help_heading = "User Report Mode")]
    pub no_diffs: bool,

    /// Show events from this date forward [default: no lower bound; GitHub keeps ≤ 90 days]
    ///
    /// Accepted formats:
    ///   ISO date    2024-01-15  or  2024-01-15T00:00:00Z
    ///   Keywords    today · yesterday
    ///   Named       last week · last month · last year
    ///   Relative    30 days ago · 2 weeks ago · 1 month ago · 1 year ago
    #[arg(long, value_name = "DATE", help_heading = "User Report Mode")]
    pub since: Option<String>,

    /// Show events up to and including this date [default: no upper bound]
    ///
    /// Same formats as --since.
    #[arg(long, value_name = "DATE", help_heading = "User Report Mode")]
    pub until: Option<String>,

    /// Event types to include in the activity feed [default: all]
    ///
    /// all     — every public event (pushes, PRs, issues, stars, forks, …)
    /// commits — push events only
    #[arg(long, value_enum, default_value_t = ActivityFilter::All, help_heading = "User Report Mode")]
    pub activity: ActivityFilter,

    /// Maximum events shown in the activity feed [default: 30]
    ///
    /// Fetches up to 100 events from GitHub and applies --since/--until/--activity
    /// filters before counting toward this limit.
    #[arg(long, default_value_t = 30, help_heading = "User Report Mode")]
    pub events: usize,
}

fn after_help_text() -> &'static str {
    static TEXT: std::sync::OnceLock<String> = std::sync::OnceLock::new();
    TEXT.get_or_init(|| {
```

```rust
        let version = match option_env!("GITPRINT_GIT_BRANCH") {
            Some(branch) => branch.to_string(),
            None => format!("v{}", env!("CARGO_PKG_VERSION")),
        };
        let size_str = std::env::current_exe()
            .ok()
            .and_then(|p| std::fs::metadata(p).ok())
            .map(|m| {
                let bytes = m.len();
                let (size, unit) = if bytes >= 1_048_576 {
                    (bytes as f64 / 1_048_576.0, "MB")
                } else {
                    (bytes as f64 / 1_024.0, "KB")
                };
                format!("{size:.1} {unit}")
            })
            .unwrap_or_else(|| "unknown".to_string());
        format!("Version: {version} | Binary size: {size_str}\nSponsor: https://github.com/sponsors/izelnakri")
    })
}

#[cfg(test)]
mod tests {
    use super::*;
    use clap::Parser;

    #[test]
    fn requires_path_or_user() {
        assert!(Args::try_parse_from(["gitprint"]).is_err());
    }

    #[test]
    fn accepts_path() {
        let args = Args::parse_from(["gitprint", "."]);
        assert_eq!(args.path, Some(".".to_string()));
    }

    #[test]
    fn custom_path() {
        let args = Args::parse_from(["gitprint", "/tmp/repo"]);
        assert_eq!(args.path, Some("/tmp/repo".to_string()));
    }

    #[test]
    fn accepts_https_url() {
        let args = Args::parse_from(["gitprint", "https://github.com/user/repo"]);
        assert_eq!(args.path, Some("https://github.com/user/repo".to_string()));
    }

    #[test]
    fn accepts_ssh_url() {
        let args = Args::parse_from(["gitprint", "git@github.com:user/repo.git"]);
        assert_eq!(args.path, Some("git@github.com:user/repo.git".to_string()));
    }

    #[test]
    fn user_flag_short() {
        let args = Args::parse_from(["gitprint", "-u", "izelnakri"]);
        assert_eq!(args.user, Some("izelnakri".to_string()));
        assert_eq!(args.path, None);
    }

    #[test]
    fn user_flag_long() {
        let args = Args::parse_from(["gitprint", "--user", "torvalds"]);
        assert_eq!(args.user, Some("torvalds".to_string()));
    }

    #[test]
    fn user_flag_with_output() {
        let args = Args::parse_from(["gitprint", "-u", "alice", "-o", "alice.pdf"]);
        assert_eq!(args.user, Some("alice".to_string()));
        assert_eq!(args.output, Some(PathBuf::from("alice.pdf")));
    }

    #[test]
    fn user_report_flags_defaults() {
        let args = Args::parse_from(["gitprint", "-u", "alice"]);
```

```
        assert_eq!(args.last_repos, 5);
        assert_eq!(args.last_commits, 5);
        assert!(!args.no_diffs);
        assert_eq!(args.events, 30);
        assert!(matches!(args.activity, ActivityFilter::All));
        assert!(args.since.is_none());
        assert!(args.until.is_none());
    }

    #[test]
    fn since_until_flags() {
        let args = Args::parse_from(["gitprint", "-u", "alice", "--since", "2024-01-01"]);
        assert_eq!(args.since.as_deref(), Some("2024-01-01"));
        let args = Args::parse_from([
            "gitprint",
            "-u",
            "alice",
            "--since",
            "30 days ago",
            "--until",
            "yesterday",
        ]);
        assert_eq!(args.since.as_deref(), Some("30 days ago"));
        assert_eq!(args.until.as_deref(), Some("yesterday"));
    }

    #[test]
    fn activity_flag() {
        let args = Args::parse_from(["gitprint", "-u", "alice", "--activity", "commits"]);
        assert!(matches!(args.activity, ActivityFilter::Commits));
        let args = Args::parse_from(["gitprint", "-u", "alice", "--activity", "all"]);
        assert!(matches!(args.activity, ActivityFilter::All));
    }

    #[test]
    fn events_flag() {
        let args = Args::parse_from(["gitprint", "-u", "alice", "--events", "50"]);
        assert_eq!(args.events, 50);
    }

    #[test]
    fn user_report_flags_custom() {
        let args = Args::parse_from([
            "gitprint",
            "-u",
            "alice",
            "--last-commits",
            "3",
            "--no-diffs",
        ]);
        assert_eq!(args.last_commits, 3);
        assert!(args.no_diffs);
    }

    #[test]
    fn output_short_flag() {
        let args = Args::parse_from(["gitprint", ".", "-o", "out.pdf"]);
        assert_eq!(args.output, Some(PathBuf::from("out.pdf")));
    }

    #[test]
    fn output_long_flag() {
        let args = Args::parse_from(["gitprint", ".", "--output", "out.pdf"]);
        assert_eq!(args.output, Some(PathBuf::from("out.pdf")));
    }

    #[test]
    fn all_flags() {
        let args = Args::parse_from([
            "gitprint",
            "https://github.com/user/repo",
            "-o",
            "out.pdf",
            "--theme",
            "Solarized (dark)",
            "--font-size",
            "10",
            "--no-line-numbers",
```

```
            "--no-toc",
            "--no-file-tree",
            "--branch",
            "dev",
            "--paper-size",
            "letter",
            "--landscape",
            "--list-themes",
        ]);
        assert_eq!(args.path, Some("https://github.com/user/repo".to_string()));
        assert_eq!(args.output, Some(PathBuf::from("out.pdf")));
        assert_eq!(args.theme, "Solarized (dark)");
        assert_eq!(args.font_size, 10.0);
        assert!(args.no_line_numbers);
        assert!(args.no_toc);
        assert!(args.no_file_tree);
        assert_eq!(args.branch, Some("dev".to_string()));
        assert!(matches!(args.paper_size, PaperSize::Letter));
        assert!(args.landscape);
        assert!(args.list_themes);
    }

    #[test]
    fn list_tags_flag() {
        let args = Args::parse_from(["gitprint", ".", "--list-tags"]);
        assert!(args.list_tags);
        let args = Args::parse_from(["gitprint", "."]);
        assert!(!args.list_tags);
    }

    #[test]
    fn commit_flag() {
        let args = Args::parse_from(["gitprint", ".", "--commit", "abc1234"]);
        assert_eq!(args.commit, Some("abc1234".to_string()));
    }

    #[test]
    fn paper_size_legal() {
        let args = Args::parse_from(["gitprint", ".", "--paper-size", "legal"]);
        assert!(matches!(args.paper_size, PaperSize::Legal));
    }

    #[test]
    fn multiple_include_exclude() {
        let args = Args::parse_from([
            "gitprint",
            ".",
            "--include",
            "*.rs",
            "--include",
            "*.toml",
            "--exclude",
            "*.lock",
            "--exclude",
            "*.md",
        ]);
        assert_eq!(args.include, vec!["*.rs", "*.toml"]);
        assert_eq!(args.exclude, vec!["*.lock", "*.md"]);
    }

    #[test]
    fn font_size_custom() {
        let args = Args::parse_from(["gitprint", ".", "--font-size", "12.5"]);
        assert_eq!(args.font_size, 12.5);
    }

    #[test]
    fn preview_flag() {
        let args = Args::parse_from(["gitprint", ".", "--preview"]);
        assert!(args.preview);
        let args = Args::parse_from(["gitprint", "."]);
        assert!(!args.preview);
    }

    #[test]
    fn preview_with_user() {
        let args = Args::parse_from(["gitprint", "-u", "alice", "--preview"]);
        assert!(args.preview);
```

```
        assert_eq!(args.user, Some("alice".to_string()));
    }
}
```

src/defaults.rs 135 LOC · 3.1 KB · 2026-03-14

```rust
/// Default glob patterns excluded from PDF output (lock files, build artifacts, binaries, etc.).
pub const DEFAULT_EXCLUDES: &[&str] = &[
    // Lock files
    "package-lock.json",
    "yarn.lock",
    "pnpm-lock.yaml",
    "Cargo.lock",
    "Gemfile.lock",
    "composer.lock",
    "poetry.lock",
    "Pipfile.lock",
    "flake.lock",
    // Build output
    "node_modules/**",
    "target/**",
    "dist/**",
    "build/**",
    ".next/**",
    "__pycache__/**",
    "*.pyc",
    // VCS / IDE
    ".git/**",
    ".svn/**",
    ".idea/**",
    ".vscode/**",
    "*.swp",
    "*.swo",
    ".DS_Store",
    // Images
    "*.png",
    "*.jpg",
    "*.jpeg",
    "*.gif",
    "*.ico",
    "*.svg",
    "*.webp",
    "*.bmp",
    // Fonts
    "*.woff",
    "*.woff2",
    "*.ttf",
    "*.otf",
    "*.eot",
    // Archives / binaries
    "*.zip",
    "*.tar",
    "*.gz",
    "*.bz2",
    "*.xz",
    "*.7z",
    "*.rar",
    "*.exe",
    "*.dll",
    "*.so",
    "*.dylib",
    "*.o",
    "*.a",
    "*.class",
    "*.jar",
    "*.war",
    "*.wasm",
    // Generated / minified
    "*.min.js",
    "*.min.css",
    "*.map",
    "*.bundle.js",
    // Data
    "*.sqlite",
    "*.db",
    "*.pdf",
];

#[cfg(test)]
mod tests {
    use super::*;
    use globset::Glob;

```

```rust
    #[test]
    fn default_excludes_has_entries() {
        assert!(DEFAULT_EXCLUDES.len() > 10);
    }

    #[test]
    fn all_patterns_are_valid_globs() {
        DEFAULT_EXCLUDES.iter().for_each(|pattern| {
            Glob::new(pattern).unwrap_or_else(|e| panic!("invalid glob '{pattern}': {e}"));
        });
    }

    #[test]
    fn known_lock_files_present() {
        assert!(DEFAULT_EXCLUDES.contains(&"Cargo.lock"));
        assert!(DEFAULT_EXCLUDES.contains(&"package-lock.json"));
        assert!(DEFAULT_EXCLUDES.contains(&"yarn.lock"));
        assert!(DEFAULT_EXCLUDES.contains(&"poetry.lock"));
        assert!(DEFAULT_EXCLUDES.contains(&"flake.lock"));
    }

    #[test]
    fn known_build_dirs_present() {
        assert!(DEFAULT_EXCLUDES.contains(&"node_modules/**"));
        assert!(DEFAULT_EXCLUDES.contains(&"target/**"));
        assert!(DEFAULT_EXCLUDES.contains(&"dist/**"));
        assert!(DEFAULT_EXCLUDES.contains(&"__pycache__/**"));
    }

    #[test]
    fn known_image_extensions_present() {
        assert!(DEFAULT_EXCLUDES.contains(&"*.png"));
        assert!(DEFAULT_EXCLUDES.contains(&"*.jpg"));
        assert!(DEFAULT_EXCLUDES.contains(&"*.svg"));
    }

    #[test]
    fn known_binary_extensions_present() {
        assert!(DEFAULT_EXCLUDES.contains(&"*.exe"));
        assert!(DEFAULT_EXCLUDES.contains(&"*.wasm"));
        assert!(DEFAULT_EXCLUDES.contains(&"*.zip"));
    }

    #[test]
    fn known_generated_extensions_present() {
        assert!(DEFAULT_EXCLUDES.contains(&"*.min.js"));
        assert!(DEFAULT_EXCLUDES.contains(&"*.min.css"));
        assert!(DEFAULT_EXCLUDES.contains(&"*.map"));
    }

    #[test]
    fn vcs_and_ide_dirs_present() {
        assert!(DEFAULT_EXCLUDES.contains(&".git/**"));
        assert!(DEFAULT_EXCLUDES.contains(&".idea/**"));
        assert!(DEFAULT_EXCLUDES.contains(&".vscode/**"));
        assert!(DEFAULT_EXCLUDES.contains(&".DS_Store"));
    }
}
```

src/filter.rs

```rust
use std::path::{Path, PathBuf};

use globset::{Glob, GlobSet, GlobSetBuilder};

use crate::defaults::DEFAULT_EXCLUDES;

/// Filters file paths based on glob include/exclude patterns.
///
/// Exclude patterns always take precedence over include patterns.
/// Default excludes (lock files, binaries, build artifacts) are always applied.
pub struct FileFilter {
    include_set: Option<GlobSet>,
    exclude_set: GlobSet,
}

impl FileFilter {
    /// Creates a new `FileFilter` from glob include and exclude patterns.
    ///
    /// An empty `include_patterns` slice allows all files (subject to excludes).
    /// Default excludes (lock files, build artifacts, binaries, etc.) are always applied.
    ///
    /// # Errors
    ///
    /// Returns an error if any glob pattern is invalid.
    ///
    /// # Examples
    ///
    /// ```
    /// use gitprint::filter::FileFilter;
    /// use std::path::Path;
    ///
    /// // Include only Rust files, exclude test helpers
    /// let filter = FileFilter::new(
    ///     &["*.rs".to_string()],
    ///     &["test_*.rs".to_string()],
    /// ).unwrap();
    ///
    /// assert!(filter.should_include(Path::new("main.rs")));
    /// assert!(!filter.should_include(Path::new("test_helper.rs")));
    /// assert!(!filter.should_include(Path::new("README.md")));
    /// ```
    pub fn new(include_patterns: &[String], exclude_patterns: &[String]) -> anyhow::Result<Self> {
        let include_set = if include_patterns.is_empty() {
            None
        } else {
            let set = include_patterns
                .iter()
                .try_fold(GlobSetBuilder::new(), |mut b, p| {
                    b.add(
                        Glob::new(p)
                            .map_err(|e| anyhow::anyhow!("invalid glob pattern '{p}': {e}"))?,
                    );
                    Ok::<_, anyhow::Error>(b)
                })?
                .build()
                .map_err(|e| anyhow::anyhow!("failed to build glob set: {e}"))?;
            Some(set)
        };

        let exclude_set = DEFAULT_EXCLUDES
            .iter()
            .map(|p| Glob::new(p).unwrap())
            .chain(
                exclude_patterns
                    .iter()
                    .map(|p| {
                        Glob::new(p).map_err(|e| anyhow::anyhow!("invalid glob pattern '{p}': {e}"))
                    })
                    .collect::<anyhow::Result<Vec<_>>>()?
                    .into_iter(),
            )
            .fold(GlobSetBuilder::new(), |mut b, g| {
                b.add(g);
                b
            })
            .build()
            .map_err(|e| anyhow::anyhow!("failed to build glob set: {e}"))?;
```

```rust

        Ok(Self {
            include_set,
            exclude_set,
        })
    }

    /// Returns `true` if `path` should be included given the configured patterns.
    ///
    /// Exclude patterns always win over include patterns.
    ///
    /// # Examples
    ///
    /// ```
    /// use gitprint::filter::FileFilter;
    /// use std::path::Path;
    ///
    /// let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
    /// assert!(filter.should_include(Path::new("src/lib.rs")));
    /// assert!(!filter.should_include(Path::new("Cargo.toml")));
    /// assert!(!filter.should_include(Path::new("Cargo.lock"))); // default exclude
    /// ```
    pub fn should_include(&self, path: &Path) -> bool {
        if self.exclude_set.is_match(path) {
            return false;
        }
        self.include_set
            .as_ref()
            .is_none_or(|set| set.is_match(path))
    }

    /// Filters a list of paths, retaining only those that pass `should_include`.
    ///
    /// # Examples
    ///
    /// ```
    /// use gitprint::filter::FileFilter;
    /// use std::path::PathBuf;
    ///
    /// let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
    /// let paths = vec![
    ///     PathBuf::from("main.rs"),
    ///     PathBuf::from("README.md"),
    ///     PathBuf::from("lib.rs"),
    /// ];
    /// let kept: Vec<_> = filter.filter_paths(paths).collect();
    /// assert_eq!(kept, vec![PathBuf::from("main.rs"), PathBuf::from("lib.rs")]);
    /// ```
    pub fn filter_paths(&self, paths: Vec<PathBuf>) -> impl Iterator<Item = PathBuf> + '_ {
        paths.into_iter().filter(|p| self.should_include(p))
    }
}

/// Returns `true` if the content appears to be a binary file.
///
/// Detection is based on the presence of non-text byte sequences (e.g. null bytes).
///
/// # Examples
///
/// ```
/// use gitprint::filter::is_binary;
///
/// assert!(is_binary(b"hello\x00world")); // null byte → binary
/// assert!(!is_binary(b"fn main() {}")); // valid UTF-8 → not binary
/// assert!(!is_binary(b""));
/// ```
pub fn is_binary(content: &[u8]) -> bool {
    content.iter().take(8000).any(|&b| b == 0)
}

/// Returns `true` if the content appears to be minified.
///
/// A file is considered minified when any of its first 5 lines exceeds 500 characters,
/// which is characteristic of bundled or minified JavaScript/CSS.
///
/// # Examples
///
/// ```
```

```rust
/// use gitprint::filter::is_minified;
///
/// assert!(is_minified(&"x".repeat(501)));   // single very long line
/// assert!(!is_minified("fn main() {\n    println!(\"hello\");\n}\n"));
/// assert!(!is_minified(""));
/// ```
pub fn is_minified(content: &str) -> bool {
    content.lines().take(5).any(|line| line.len() > 500)
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn default_excludes_applied() {
        let filter = FileFilter::new(&[], &[]).unwrap();
        assert!(!filter.should_include(Path::new("Cargo.lock")));
        assert!(!filter.should_include(Path::new("node_modules/foo.js")));
        assert!(!filter.should_include(Path::new("image.png")));
        assert!(!filter.should_include(Path::new("target/debug/binary")));
        assert!(!filter.should_include(Path::new(".git/HEAD")));
        assert!(!filter.should_include(Path::new("bundle.min.js")));
    }

    #[test]
    fn custom_exclude() {
        let filter = FileFilter::new(&[], &["*.md".to_string()]).unwrap();
        assert!(!filter.should_include(Path::new("README.md")));
        assert!(!filter.should_include(Path::new("docs/GUIDE.md")));
        assert!(filter.should_include(Path::new("main.rs")));
    }

    #[test]
    fn include_only() {
        let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
        assert!(filter.should_include(Path::new("main.rs")));
        assert!(filter.should_include(Path::new("src/lib.rs")));
        assert!(!filter.should_include(Path::new("README.md")));
        assert!(!filter.should_include(Path::new("Cargo.toml")));
    }

    #[test]
    fn include_and_exclude_interaction() {
        let filter = FileFilter::new(&["*.rs".to_string()], &["test_*.rs".to_string()]).unwrap();
        assert!(filter.should_include(Path::new("main.rs")));
        assert!(!filter.should_include(Path::new("test_helper.rs")));
    }

    #[test]
    fn empty_filter_includes_normal_files() {
        let filter = FileFilter::new(&[], &[]).unwrap();
        assert!(filter.should_include(Path::new("src/main.rs")));
        assert!(filter.should_include(Path::new("Cargo.toml")));
        assert!(filter.should_include(Path::new("README.md")));
    }

    #[test]
    fn multiple_include_patterns() {
        let filter = FileFilter::new(&["*.rs".to_string(), "*.toml".to_string()], &[]).unwrap();
        assert!(filter.should_include(Path::new("main.rs")));
        assert!(filter.should_include(Path::new("Cargo.toml")));
        assert!(!filter.should_include(Path::new("README.md")));
    }

    #[test]
    fn multiple_exclude_patterns() {
        let filter = FileFilter::new(&[], &["*.md".to_string(), "*.txt".to_string()]).unwrap();
        assert!(!filter.should_include(Path::new("README.md")));
        assert!(!filter.should_include(Path::new("notes.txt")));
        assert!(filter.should_include(Path::new("main.rs")));
    }

    #[test]
    fn exclude_takes_precedence_over_include() {
        let filter = FileFilter::new(&["*.rs".to_string()], &["main.rs".to_string()]).unwrap();
        assert!(!filter.should_include(Path::new("main.rs")));
        assert!(filter.should_include(Path::new("lib.rs")));
```

```
    }

    #[test]
    fn filter_paths_works() {
        let filter = FileFilter::new(&["*.rs".to_string()], &[]).unwrap();
        let paths = vec![
            PathBuf::from("main.rs"),
            PathBuf::from("README.md"),
            PathBuf::from("lib.rs"),
        ];
        let filtered: Vec<_> = filter.filter_paths(paths).collect();
        assert_eq!(
            filtered,
            vec![PathBuf::from("main.rs"), PathBuf::from("lib.rs")]
        );
    }

    #[test]
    fn filter_paths_empty_input() {
        let filter = FileFilter::new(&[], &[]).unwrap();
        let filtered: Vec<_> = filter.filter_paths(vec![]).collect();
        assert!(filtered.is_empty());
    }

    #[test]
    fn is_binary_with_null_bytes() {
        let content = b"hello\x00world";
        assert!(is_binary(content));
    }

    #[test]
    fn is_binary_with_text() {
        let content = b"fn main() { println!(\"hello\"); }";
        assert!(!is_binary(content));
    }

    #[test]
    fn is_binary_with_empty() {
        assert!(!is_binary(b""));
    }

    #[test]
    fn is_binary_with_utf8() {
        assert!(!is_binary("□ □ □ □ □ □ □ ".as_bytes()));
    }

    #[test]
    fn is_minified_with_long_line() {
        let long_line = "a".repeat(501);
        assert!(is_minified(&long_line));
    }

    #[test]
    fn is_minified_with_normal_content() {
        assert!(!is_minified("fn main() {\n    println!(\"hi\");\n}\n"));
    }

    #[test]
    fn is_minified_long_line_after_fifth() {
        let mut content = "short\n".repeat(5);
        content.push_str(&"a".repeat(501));
        assert!(!is_minified(&content));
    }

    #[test]
    fn is_minified_exactly_500_chars() {
        let line = "a".repeat(500);
        assert!(!is_minified(&line));
    }

    #[test]
    fn is_minified_empty() {
        assert!(!is_minified(""));
    }

    #[test]
    fn is_minified_long_line_on_line_3() {
        let content = format!("short\nshort\n{}\nshort\nshort\n", "a".repeat(501));
```

```
        assert!(is_minified(&content));
    }

    #[test]
    fn invalid_include_glob_returns_error() {
        let result = FileFilter::new(&["[invalid".to_string()], &[]);
        assert!(result.is_err());
    }

    #[test]
    fn invalid_exclude_glob_returns_error() {
        let result = FileFilter::new(&[], &["[invalid".to_string()]);
        assert!(result.is_err());
    }
}
```

src/git.rs 858 LOC · 27.6 KB · 2026-03-14

```rust
use std::collections::HashMap;
use std::future::Future;
use std::path::{Path, PathBuf};
use std::pin::Pin;
use std::sync::Arc;
use std::time::UNIX_EPOCH;

use anyhow::bail;
use tokio::process::Command;

use crate::types::{Config, RepoMetadata};

/// Returns `true` if `s` looks like a remote git URL.
///
/// Recognised schemes: `https://`, `http://`, `git://`, `ssh://`,
/// and SCP-style `git@host:path` used by GitHub/GitLab.
pub fn is_remote_url(s: &str) -> bool {
    s.starts_with("https://")
        || s.starts_with("http://")
        || s.starts_with("git://")
        || s.starts_with("ssh://")
        || (s.contains('@') && s.contains(':') && !s.starts_with('/'))
}

/// Extracts the repository name from a remote URL.
///
/// `https://github.com/user/repo.git` → `"repo"`
/// `git@github.com:user/repo`         → `"repo"`
pub fn repo_name_from_url(url: &str) -> String {
    url.split(['/', ':'])
        .next_back()
        .unwrap_or("repo")
        .trim_end_matches(".git")
        .to_string()
}

/// A temporary directory that deletes itself on drop.
pub struct TempCloneDir(PathBuf);

impl TempCloneDir {
    /// Creates (or reuses) a deterministically-named temp dir based on the URL,
    /// branch, and commit so repeated invocations for the same target don't
    /// accumulate stale copies in `/tmp`.
    pub async fn for_url(
        url: &str,
        branch: Option<&str>,
        commit: Option<&str>,
    ) -> anyhow::Result<Self> {
        use std::collections::hash_map::DefaultHasher;
        use std::hash::{Hash, Hasher};
        let mut h = DefaultHasher::new();
        url.hash(&mut h);
        branch.hash(&mut h);
        commit.hash(&mut h);
        let dir = std::env::temp_dir().join(format!("gitprint-{:016x}", h.finish()));
        tokio::fs::create_dir_all(&dir).await?;
        Ok(Self(dir))
    }

    /// Returns the path to the temporary clone directory.
    pub fn path(&self) -> &Path {
        &self.0
    }
}

impl Drop for TempCloneDir {
    fn drop(&mut self) {
        // Drop is synchronous by design — tokio async cannot be used here.
        let _ = std::fs::remove_dir_all(&self.0);
    }
}

/// Clones a remote git repository into `dest`.
///
/// Uses `--depth=1` (shallow) for speed unless `commit` is specified, in which
/// case a full clone is required to access arbitrary history.
pub async fn clone_repo(
```

```rust
    url: &str,
    dest: &Path,
    branch: Option<&str>,
    commit: Option<&str>,
) -> anyhow::Result<()> {
    let mut cmd = Command::new("git");
    cmd.arg("clone");

    if commit.is_none() {
        cmd.arg("--depth=1");
        if let Some(b) = branch {
            cmd.args(["--branch", b, "--single-branch"]);
        }
    } else if let Some(b) = branch {
        cmd.args(["--branch", b]);
    }

    let status = cmd
        .arg(url)
        .arg(dest)
        .stderr(std::process::Stdio::inherit())
        .status()
        .await
        .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;

    if !status.success() {
        bail!("git clone failed for {url}");
    }
    Ok(())
}

async fn run_git(repo_path: &Path, args: &[&str]) -> anyhow::Result<String> {
    let output = Command::new("git")
        .args(["-C", &repo_path.to_string_lossy()])
        .args(args)
        .output()
        .await
        .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;

    if !output.status.success() {
        let stderr = String::from_utf8_lossy(&output.stderr);
        bail!("{}", stderr.trim());
    }

    Ok(String::from_utf8(output.stdout)
        .unwrap_or_else(|e| String::from_utf8_lossy(e.as_bytes()).into_owned()))
}

/// Describes what the user-supplied path resolves to.
#[derive(Debug)]
pub struct RepoInfo {
    /// Git repo root (git mode) or canonical directory path (plain-dir mode).
    pub root: PathBuf,
    /// Whether `root` is inside a git repository.
    pub is_git: bool,
    /// Subdirectory scope within the git repo (relative to `root`).
    /// Only set when the user supplied a strict subdirectory of the repo root.
    pub scope: Option<PathBuf>,
    /// When the user supplied a single file, its path relative to `root`.
    pub single_file: Option<PathBuf>,
}

/// Resolves a user-supplied path into a [`RepoInfo`].
///
/// Handles four cases:
///
/// - File inside a git repo → `single_file` is set, `root` is the repo root.
/// - Subdirectory inside a git repo → `scope` is set relative to `root`.
/// - Git repo root → `root` is the repo root, no scope.
/// - Plain directory or file outside git → `is_git` is `false`.
///
/// # Errors
///
/// Returns an error if the path does not exist.
///
/// # Examples
///
/// ```ignore
```

```rust
/// use gitprint::git::verify_repo;
/// use std::path::Path;
///
/// let info = verify_repo(Path::new(".")).await.unwrap();
/// println!("repo root: {}", info.root.display());
/// println!("is git: {}", info.is_git);
/// ```
pub async fn verify_repo(path: &Path) -> anyhow::Result<RepoInfo> {
    // Use async canonicalize to avoid blocking tokio worker threads.
    let canonical = tokio::fs::canonicalize(path)
        .await
        .map_err(|_| anyhow::anyhow!("{}: path not found", path.display()))?;

    // Fetch metadata once (async stat) and reuse is_file/is_dir throughout —
    // avoids multiple blocking stat() calls on the same already-resolved path.
    let meta = tokio::fs::metadata(&canonical)
        .await
        .map_err(|_| anyhow::anyhow!("{}: cannot stat path", canonical.display()))?;
    let is_file = meta.is_file();
    let is_dir = meta.is_dir();

    // Git must be invoked from a directory; use parent when the path is a file.
    let git_dir = if is_file {
        canonical
            .parent()
            .ok_or_else(|| anyhow::anyhow!("file has no parent directory"))?
            .to_path_buf()
    } else {
        canonical.clone()
    };

    let output = Command::new("git")
        .args(["-C", &git_dir.to_string_lossy()])
        .args(["rev-parse", "--show-toplevel"])
        .output()
        .await
        .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;

    if output.status.success() {
        let root = PathBuf::from(String::from_utf8_lossy(&output.stdout).trim().to_string());

        if is_file {
            let rel = canonical
                .strip_prefix(&root)
                .map_err(|_| anyhow::anyhow!("file is outside the git repository"))?
                .to_path_buf();
            return Ok(RepoInfo {
                root,
                is_git: true,
                scope: None,
                single_file: Some(rel),
            });
        }

        let scope = (canonical != root)
            .then(|| canonical.strip_prefix(&root).ok().map(|p| p.to_path_buf()))
            .flatten();
        return Ok(RepoInfo {
            root,
            is_git: true,
            scope,
            single_file: None,
        });
    }

    // Not inside a git repo.
    if is_file {
        let parent = canonical
            .parent()
            .ok_or_else(|| anyhow::anyhow!("file has no parent directory"))?
            .to_path_buf();
        return Ok(RepoInfo {
            root: parent,
            is_git: false,
            scope: None,
            single_file: Some(PathBuf::from(canonical.file_name().unwrap())),
        });
    }
```

```rust

    if is_dir {
        return Ok(RepoInfo {
            root: canonical,
            is_git: false,
            scope: None,
            single_file: None,
        });
    }

    bail!(
        "{}: not a git repository, directory, or file",
        path.display()
    )
}

/// Fetches repository metadata: branch, last commit hash/date/message, and name.
///
/// For non-git directories, returns a `RepoMetadata` with empty git fields.
/// Branch detection and commit log are fetched concurrently.
///
/// # Errors
///
/// Returns an error if the git command fails (git repos only).
pub async fn get_metadata(
    repo_path: &Path,
    config: &Config,
    is_git: bool,
    scope: Option<&Path>,
) -> anyhow::Result<RepoMetadata> {
    let base = repo_path
        .file_name()
        .map(|n| n.to_string_lossy().to_string())
        .unwrap_or_else(|| "unknown".to_string());
    let name = match scope {
        Some(s) => format!("{}/{}", base, s.display()),
        None => base,
    };

    if !is_git {
        return Ok(RepoMetadata {
            name,
            branch: String::new(),
            commit_hash: String::new(),
            commit_hash_short: String::new(),
            commit_date: String::new(),
            commit_message: String::new(),
            commit_author: String::new(),
            commit_author_email: String::new(),
            file_count: 0,
            total_lines: 0,
            fs_owner: None,
            fs_group: None,
            generated_at: String::new(),
            repo_size: String::new(),
            fs_size: String::new(),
            detected_remote_url: None,
            repo_absolute_path: None,
        });
    }

    let rev = match (&config.commit, &config.branch) {
        (Some(c), _) => c.clone(),
        (_, Some(b)) => b.clone(),
        _ => "HEAD".to_string(),
    };

    // Run branch detection, commit log, and remote URL detection in parallel.
    // Format: hash, date, subject, author name, author email (one per line, %n separated).
    let log_args = ["log", "-1", "--format=%H%n%ci%n%s%n%an%n%ae", &rev];
    let (branch, log_output, detected_remote_url) = tokio::join!(
        async {
            match &config.branch {
                Some(b) => b.clone(),
                None => run_git(repo_path, &["rev-parse", "--abbrev-ref", "HEAD"])
                    .await
                    .map(|s| s.trim().to_string())
                    .unwrap_or_else(|_| "detached".to_string()),
```

```
            }
        },
        run_git(repo_path, &log_args),
        git_remote_url(repo_path),
    );
    let log_output = log_output?;

    let mut lines = log_output.trim().lines();
    let commit_hash = lines.next().unwrap_or("").to_string();
    let commit_hash_short = commit_hash[..7.min(commit_hash.len())].to_string();
    let commit_date = lines.next().unwrap_or("").to_string();
    // Remaining: subject lines, then author name, then author email (last two lines).
    let remaining: Vec<&str> = lines.collect();
    let (commit_message, commit_author, commit_author_email) = match remaining.as_slice() {
        [] => (String::new(), String::new(), String::new()),
        [.., author, email] => {
            let subject_lines = &remaining[..remaining.len().saturating_sub(2)];
            (
                subject_lines.join("\n"),
                author.to_string(),
                email.to_string(),
            )
        }
        [author] => (String::new(), author.to_string(), String::new()),
    };

    Ok(RepoMetadata {
        name,
        branch,
        commit_hash,
        commit_hash_short,
        commit_date,
        commit_message,
        commit_author,
        commit_author_email,
        file_count: 0,
        total_lines: 0,
        fs_owner: None,
        fs_group: None,
        generated_at: String::new(),
        repo_size: String::new(),
        fs_size: String::new(),
        detected_remote_url,
        repo_absolute_path: None,
    })
}

/// Lists all files to be included in the PDF.
///
/// In git mode: uses `git ls-files` (working tree) or `git ls-tree` (specific
/// branch/commit). In plain-directory mode: recursively walks the filesystem.
///
/// # Errors
///
/// Returns an error if the git command or directory walk fails.
pub async fn list_tracked_files(
    repo_path: &Path,
    config: &Config,
    is_git: bool,
    scope: Option<&Path>,
) -> anyhow::Result<Vec<PathBuf>> {
    if !is_git {
        return walk_files_async(repo_path.to_path_buf()).await;
    }

    let scope_str = scope.and_then(|p| p.to_str());
    let output = match (&config.commit, &config.branch) {
        (Some(commit), _) => match scope_str {
            Some(s) => {
                run_git(
                    repo_path,
                    &["ls-tree", "-r", "--name-only", commit, "--", s],
                )
                .await?
            }
            None => run_git(repo_path, &["ls-tree", "-r", "--name-only", commit]).await?,
        },
        (_, Some(branch)) => match scope_str {
```

```
            Some(s) => {
                run_git(
                    repo_path,
                    &["ls-tree", "-r", "--name-only", branch, "--", s],
                )
                .await?
            }
            None => run_git(repo_path, &["ls-tree", "-r", "--name-only", branch]).await?,
        },
        _ => match scope_str {
            Some(s) => run_git(repo_path, &["ls-files", "--", s]).await?,
            None => run_git(repo_path, &["ls-files"]).await?,
        },
    };

    Ok(output
        .lines()
        .filter(|l| !l.is_empty())
        .map(PathBuf::from)
        .collect())
}

/// Returns a map of file path → last modified date (YYYY-MM-DD).
/// In git mode: parsed from `git log`. In directory mode: from filesystem mtime.
pub async fn file_last_modified_dates(
    repo_path: &Path,
    config: &Config,
    is_git: bool,
    scope: Option<&Path>,
) -> anyhow::Result<HashMap<PathBuf, String>> {
    if !is_git {
        return walk_dates_async(repo_path.to_path_buf()).await;
    }

    let rev = match (&config.commit, &config.branch) {
        (Some(c), _) => c.clone(),
        (_, Some(b)) => b.clone(),
        _ => "HEAD".to_string(),
    };

    let scope_str = scope.and_then(|p| p.to_str());
    let output = match scope_str {
        Some(s) => {
            run_git(
                repo_path,
                &["log", "--format=COMMIT:%ci", "--name-only", &rev, "--", s],
            )
            .await?
        }
        None => {
            run_git(
                repo_path,
                &["log", "--format=COMMIT:%ci", "--name-only", &rev],
            )
            .await?
        }
    };

    let mut map = HashMap::new();
    let mut current_date = String::new();

    output.lines().for_each(|line| {
        if let Some(date_str) = line.strip_prefix("COMMIT:") {
            current_date = date_str.chars().take(10).collect();
        } else if !line.is_empty() && !current_date.is_empty() {
            map.entry(PathBuf::from(line))
                .or_insert_with(|| current_date.clone());
        }
    });

    Ok(map)
}

/// Returns the last-modified date (YYYY-MM-DD) for a single file.
/// In git mode: from `git log`. In plain mode: from filesystem mtime.
pub async fn file_last_modified(root: &Path, file: &Path, config: &Config, is_git: bool) -> String {
    if is_git {
        let rev = config
```

```rust
            .commit
            .as_deref()
            .or(config.branch.as_deref())
            .unwrap_or("HEAD");
        let file_str = file.to_string_lossy();
        run_git(
            root,
            &["log", "-1", "--format=%ci", rev, "--", file_str.as_ref()],
        )
        .await
        .ok()
        .map(|s| s.trim().chars().take(10).collect())
        .unwrap_or_default()
    } else {
        tokio::fs::metadata(root.join(file))
            .await
            .ok()
            .and_then(|m| m.modified().ok())
            .map(|t| {
                let secs = t.duration_since(UNIX_EPOCH).unwrap_or_default().as_secs();
                let (y, m, d) = unix_secs_to_ymd(secs);
                format!("{y:04}-{m:02}-{d:02}")
            })
            .unwrap_or_default()
    }
}

/// Reads the content of a single file, using `git show` for a specific revision or plain I/O otherwise.
pub async fn read_file_content(
    repo_path: &Path,
    file_path: &Path,
    config: &Config,
) -> anyhow::Result<String> {
    let rev = config.commit.as_deref().or(config.branch.as_deref());
    match rev {
        Some(rev) => {
            let spec = format!("{rev}:{}", file_path.display());
            run_git(repo_path, &["show", &spec]).await
        }
        None => tokio::fs::read_to_string(repo_path.join(file_path))
            .await
            .map_err(Into::into),
    }
}

// ── Private helpers for plain-directory mode ──────────────────────────

/// Converts Unix timestamp (seconds since epoch) to (year, month, day).
/// Uses Howard Hinnant's date algorithm.
fn unix_secs_to_ymd(secs: u64) -> (u32, u32, u32) {
    let z = (secs / 86400) as i64 + 719_468;
    let era = if z >= 0 { z } else { z - 146_096 } / 146_097;
    let doe = (z - era * 146_097) as u32;
    let yoe = (doe - doe / 1460 + doe / 36524 - doe / 146096) / 365;
    let y = yoe as i64 + era * 400;
    let doy = doe - (365 * yoe + yoe / 4 - yoe / 100);
    let mp = (5 * doy + 2) / 153;
    let d = doy - (153 * mp + 2) / 5 + 1;
    let m = if mp < 10 { mp + 3 } else { mp - 9 };
    let y = if m <= 2 { y + 1 } else { y };
    (y as u32, m, d)
}

/// Recursive async walk returning all file paths relative to `root`.
/// Each directory immediately spawns tasks for its subdirectories — no
/// level-by-level BFS barriers, maximum concurrency throughout the tree.
fn walk_files_inner(
    root: Arc<PathBuf>,
    dir: PathBuf,
) -> Pin<Box<dyn Future<Output = anyhow::Result<Vec<PathBuf>>> + Send>> {
    Box::pin(async move {
        let mut rd = tokio::fs::read_dir(&dir).await?;
        let mut files: Vec<PathBuf> = Vec::new();
        let mut set: tokio::task::JoinSet<anyhow::Result<Vec<PathBuf>>> =
            tokio::task::JoinSet::new();

        while let Some(entry) = rd.next_entry().await? {
            let ft = entry.file_type().await?;
```

```rust
            if ft.is_dir() {
                set.spawn(walk_files_inner(Arc::clone(&root), entry.path()));
            } else if ft.is_file()
                && let Ok(rel) = entry.path().strip_prefix(root.as_ref())
            {
                files.push(rel.to_path_buf());
            }
        }

        set.join_all()
            .await
            .into_iter()
            .try_for_each(|res| res.map(|sub| files.extend(sub)))?;

        Ok(files)
    })
}

async fn walk_files_async(root: PathBuf) -> anyhow::Result<Vec<PathBuf>> {
    walk_files_inner(Arc::new(root.clone()), root).await
}

/// Walk the tree (via `walk_files_async`) then fetch all file mtimes concurrently.
async fn walk_dates_async(root: PathBuf) -> anyhow::Result<HashMap<PathBuf, String>> {
    let files = walk_files_async(root.clone()).await?;
    let mut set: tokio::task::JoinSet<Option<(PathBuf, String)>> = tokio::task::JoinSet::new();

    files.into_iter().for_each(|rel| {
        let abs = root.join(&rel);
        set.spawn(async move {
            let date = tokio::fs::metadata(&abs)
                .await
                .ok()
                .and_then(|m| m.modified().ok())
                .map(|t| {
                    let secs = t.duration_since(UNIX_EPOCH).unwrap_or_default().as_secs();
                    let (y, m, d) = unix_secs_to_ymd(secs);
                    format!("{y:04}-{m:02}-{d:02}")
                })?;
            Some((rel, date))
        });
    });

    Ok(set.join_all().await.into_iter().flatten().collect())
}

/// Returns the filesystem owner username and group name for `path`.
///
/// Tries GNU `stat -c "%U\n%G"` (Linux/coreutils) then BSD `stat -f "%Su\n%Sg"` (macOS).
/// Returns `(None, None)` if both fail or the path is inaccessible.
pub async fn fs_owner_group(path: &Path) -> (Option<String>, Option<String>) {
    for args in [
        &["-c", "%U\n%G"][..],   // GNU stat (Linux)
        &["-f", "%Su\n%Sg"][..], // BSD stat (macOS)
    ] {
        if let Ok(out) = Command::new("stat").args(args).arg(path).output().await {
            if out.status.success() {
                let text = String::from_utf8_lossy(&out.stdout);
                let mut lines = text.trim().lines();
                return (
                    lines.next().filter(|s| !s.is_empty()).map(str::to_string),
                    lines.next().filter(|s| !s.is_empty()).map(str::to_string),
                );
            }
        }
    }
    (None, None)
}

/// Formats a byte count as a human-readable string using binary prefixes.
fn format_bytes(bytes: u64) -> String {
    if bytes < 1_024 {
        format!("{bytes} B")
    } else if bytes < 1_048_576 {
        format!("{:.1} KB", bytes as f64 / 1_024.0)
    } else if bytes < 1_073_741_824 {
        format!("{:.1} MB", bytes as f64 / 1_048_576.0)
    } else {
```

```rust
        format!("{:.1} GB", bytes as f64 / 1_073_741_824.0)
    }
}

/// Recursive half of `fs_dir_size`: sums the byte sizes of all regular files under `dir`.
fn sum_dir_bytes(dir: PathBuf) -> Pin<Box<dyn Future<Output = u64> + Send>> {
    Box::pin(async move {
        let mut rd = match tokio::fs::read_dir(&dir).await {
            Ok(rd) => rd,
            Err(_) => return 0,
        };
        let mut total = 0u64;
        let mut set: tokio::task::JoinSet<u64> = tokio::task::JoinSet::new();
        while let Ok(Some(entry)) = rd.next_entry().await {
            let Ok(ft) = entry.file_type().await else {
                continue;
            };
            if ft.is_dir() {
                set.spawn(sum_dir_bytes(entry.path()));
            } else if ft.is_file() {
                if let Ok(m) = entry.metadata().await {
                    total += m.len();
                }
            }
        }
        total += set.join_all().await.into_iter().sum::<u64>();
        total
    })
}

/// Returns the total size of all files under `path` as a human-readable string (e.g. `"4.2 MB"`).
///
/// Recursively sums actual file content bytes (not disk-block allocations) using
/// async I/O — no subprocess, no platform-specific tools, consistent formatting.
/// Returns an empty string if `path` is inaccessible or contains no files.
pub async fn fs_dir_size(path: &Path) -> String {
    let total = sum_dir_bytes(path.to_path_buf()).await;
    if total == 0 {
        String::new()
    } else {
        format_bytes(total)
    }
}

/// Returns the total size of all git-tracked files as a human-readable string (e.g. `"3.8 MB"`).
///
/// Uses `git ls-tree -r -l` to sum blob sizes from the object database — this
/// reflects actual tracked content without `.git/` overhead.
/// Falls back to an empty string on failure.
pub async fn git_tracked_size(repo_path: &Path, config: &Config) -> String {
    let rev = config
        .commit
        .as_deref()
        .or(config.branch.as_deref())
        .unwrap_or("HEAD");
    let output = run_git(repo_path, &["ls-tree", "-r", "-l", rev])
        .await
        .unwrap_or_default();
    let total_bytes: u64 = output
        .lines()
        .filter_map(|line| line.split_whitespace().nth(3)?.parse::<u64>().ok())
        .sum();
    if total_bytes == 0 {
        return String::new();
    }
    format_bytes(total_bytes)
}

/// Normalizes a git remote URL to an `https://` URL.
///
/// Handles SCP-style (`git@github.com:user/repo`) and `ssh://` URLs, converting
/// them to their `https://` equivalents so they can be used in clickable links.
/// Plain `https://` or `http://` URLs are returned unchanged.
fn normalize_to_https(url: &str) -> String {
    if url.starts_with("https://") || url.starts_with("http://") {
        return url.to_string();
    } else if let Some(rest) = url.strip_prefix("git@") {
        // SCP-style: git@github.com:user/repo.git
```

```rust
        if let Some(colon_pos) = rest.find(':') {
            return format!("https://{}/{}", &rest[..colon_pos], &rest[colon_pos + 1..]);
        }
    } else if let Some(rest) = url.strip_prefix("ssh://git@") {
        return format!("https://{rest}");
    } else if let Some(rest) = url.strip_prefix("ssh://") {
        return format!("https://{rest}");
    }
    url.to_string()
}

/// Fetches all tag refs from the remote without downloading full history.
///
/// Needed after a `--depth=1` clone, which only fetches the tag (if any)
/// pointing at the cloned commit — other tags are absent until this runs.
pub async fn fetch_tags(repo_path: &Path) -> anyhow::Result<()> {
    let status = Command::new("git")
        .args(["fetch", "--tags", "--depth=1"])
        .current_dir(repo_path)
        .status()
        .await
        .map_err(|e| anyhow::anyhow!("failed to run git: {e}"))?;
    if !status.success() {
        bail!("git fetch --tags failed");
    }
    Ok(())
}

/// Lists all version tags in a repository, sorted newest first.
///
/// Uses `git tag --list --sort=-version:refname` which sorts by semver-aware
/// descending order so `v1.10.0` sorts before `v1.9.0`.
/// Returns an empty Vec if there are no tags or the path is not a git repo.
pub async fn list_repo_tags(repo_path: &Path) -> Vec<String> {
    run_git(repo_path, &["tag", "--list", "--sort=-version:refname"])
        .await
        .unwrap_or_default()
        .lines()
        .filter(|l| !l.is_empty())
        .map(str::to_string)
        .collect()
}

/// Returns the remote URL for `origin`, if one is configured.
///
/// Runs `git remote get-url origin` — if the repo has no remote or the command
/// fails, returns `None`. SCP-style and ssh:// URLs are normalized to https://.
pub async fn git_remote_url(repo_path: &Path) -> Option<String> {
    run_git(repo_path, &["remote", "get-url", "origin"])
        .await
        .ok()
        .map(|s| normalize_to_https(s.trim()))
        .filter(|s| !s.is_empty())
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn normalize_https_passthrough() {
        assert_eq!(
            normalize_to_https("https://github.com/user/repo.git"),
            "https://github.com/user/repo.git"
        );
        assert_eq!(
            normalize_to_https("http://example.com/repo"),
            "http://example.com/repo"
        );
    }

    #[test]
    fn normalize_scp_style() {
        assert_eq!(
            normalize_to_https("git@github.com:user/repo.git"),
            "https://github.com/user/repo.git"
        );
        assert_eq!(
```

```rust
            normalize_to_https("git@gitlab.com:org/project"),
            "https://gitlab.com/org/project"
        );
    }

    #[test]
    fn normalize_ssh_git_at_style() {
        assert_eq!(
            normalize_to_https("ssh://git@github.com/user/repo.git"),
            "https://github.com/user/repo.git"
        );
    }

    #[test]
    fn normalize_ssh_style() {
        assert_eq!(
            normalize_to_https("ssh://github.com/user/repo.git"),
            "https://github.com/user/repo.git"
        );
    }

    #[test]
    fn is_remote_url_https() {
        assert!(is_remote_url("https://github.com/user/repo"));
        assert!(is_remote_url("https://github.com/user/repo.git"));
        assert!(is_remote_url("http://example.com/repo.git"));
    }

    #[test]
    fn is_remote_url_git_schemes() {
        assert!(is_remote_url("git://github.com/user/repo.git"));
        assert!(is_remote_url("ssh://git@github.com/user/repo.git"));
    }

    #[test]
    fn is_remote_url_scp_style() {
        assert!(is_remote_url("git@github.com:user/repo.git"));
        assert!(is_remote_url("git@gitlab.com:org/repo"));
    }

    #[test]
    fn is_remote_url_rejects_local() {
        assert!(!is_remote_url("."));
        assert!(!is_remote_url("/home/user/repo"));
        assert!(!is_remote_url("relative/path"));
        assert!(!is_remote_url("src/main.rs"));
    }

    #[test]
    fn repo_name_from_url_https() {
        assert_eq!(
            repo_name_from_url("https://github.com/user/repo.git"),
            "repo"
        );
        assert_eq!(repo_name_from_url("https://github.com/user/repo"), "repo");
    }

    #[test]
    fn repo_name_from_url_scp() {
        assert_eq!(repo_name_from_url("git@github.com:user/repo.git"), "repo");
        assert_eq!(
            repo_name_from_url("git@gitlab.com:org/myproject"),
            "myproject"
        );
    }

    #[tokio::test]
    async fn temp_clone_dir_creates_and_cleans_up() {
        let path = {
            let t = TempCloneDir::for_url("https://example.com/repo", None, None)
                .await
                .unwrap();
            let p = t.path().to_path_buf();
            assert!(p.exists());
            p
        };
        assert!(!path.exists());
    }
```

```
858 }
```

src/github.rs 435 LOC · 14.1 KB · 2026-03-14

```rust
//! GitHub REST API v3 client.
//!
//! All functions operate on public data and work without authentication.
//! Set `GITHUB_TOKEN` in the environment for higher rate limits (5 000/hr vs 60/hr)
//! and access to private repositories.

use anyhow::{Context, bail};
use serde::Deserialize;

const API_BASE: &str = "https://api.github.com";
const VERSION: &str = env!("CARGO_PKG_VERSION");

// ── Response types ──────────────────────────────────────────────────────────

/// GitHub user public profile returned by `GET /users/{username}`.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct GitHubUser {
    pub login: String,
    pub name: Option<String>,
    pub bio: Option<String>,
    pub location: Option<String>,
    pub company: Option<String>,
    pub blog: Option<String>,
    pub email: Option<String>,
    pub public_repos: u64,
    pub followers: u64,
    pub following: u64,
    pub created_at: String,
    pub html_url: String,
}

/// A GitHub repository as returned by the repos and search APIs.
#[allow(missing_docs)]
#[derive(Debug, Deserialize, Clone)]
pub struct GitHubRepo {
    pub name: String,
    pub full_name: String,
    pub html_url: String,
    pub description: Option<String>,
    pub language: Option<String>,
    pub stargazers_count: u64,
    pub forks_count: u64,
    pub pushed_at: Option<String>,
    pub updated_at: Option<String>,
    pub fork: bool,
    #[serde(default)]
    pub open_issues_count: u64,
    #[serde(default)]
    pub size: u64, // in KB
    #[serde(default)]
    pub created_at: Option<String>,
}

/// A public GitHub event as returned by `GET /users/{username}/events/public`.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct GitHubEvent {
    #[serde(rename = "type")]
    pub kind: String,
    pub repo: EventRepo,
    pub payload: serde_json::Value,
    pub created_at: String,
}

/// The repository reference embedded in a GitHub event.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct EventRepo {
    pub name: String,
}

/// A single commit with its file patches, as returned by `GET /repos/{owner}/{repo}/commits/{sha}`.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct CommitDetail {
    pub sha: String,
```

```rust
    pub html_url: String,
    pub commit: CommitInfo,
    #[serde(default)]
    pub files: Vec<CommitFile>,
}

/// Commit metadata (message and author) embedded in a `CommitDetail`.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct CommitInfo {
    pub message: String,
    pub author: CommitAuthor,
}

/// Author name and date embedded in a `CommitInfo`.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct CommitAuthor {
    pub name: String,
    pub date: String,
}

/// A single changed file within a commit, including optional unified diff patch.
#[allow(missing_docs)]
#[derive(Debug, Deserialize)]
pub struct CommitFile {
    pub filename: String,
    pub status: String,
    pub additions: u64,
    pub deletions: u64,
    pub patch: Option<String>,
}

// ── Client helpers ──────────────────────────────────────────────────────

pub(crate) fn build_client() -> anyhow::Result<reqwest::Client> {
    reqwest::Client::builder()
        .user_agent(format!("gitprint/{VERSION}"))
        .build()
        .context("failed to build HTTP client")
}

fn auth_header(token: Option<&str>) -> Option<String> {
    token.map(|t| format!("Bearer {t}"))
}

pub(crate) async fn get_json<T: for<'de> Deserialize<'de>>(
    client: &reqwest::Client,
    url: &str,
    token: Option<&str>,
) -> anyhow::Result<T> {
    let mut req = client
        .get(url)
        .header("Accept", "application/vnd.github+json");
    if let Some(auth) = auth_header(token) {
        req = req.header("Authorization", auth);
    }
    let resp = req.send().await.with_context(|| format!("GET {url}"))?;
    let status = resp.status();
    if status == reqwest::StatusCode::NOT_FOUND {
        bail!("not found: {url}");
    }
    if status == reqwest::StatusCode::FORBIDDEN || status == reqwest::StatusCode::TOO_MANY_REQUESTS
    {
        bail!(
            "GitHub API rate limit exceeded. Set GITHUB_TOKEN to increase limits:\n  \
             export GITHUB_TOKEN=ghp_your_token_here"
        );
    }
    if !status.is_success() {
        bail!("GitHub API error {status}: {url}");
    }
    resp.json::<T>()
        .await
        .with_context(|| format!("parsing response from {url}"))
}

// ── Public API functions ────────────────────────────────────────────────
```

```rust

/// Fetch a user's public profile.
pub async fn get_user(username: &str, token: Option<&str>) -> anyhow::Result<GitHubUser> {
    let client = build_client()?;
    let url = format!("{API_BASE}/users/{username}");
    get_json::<GitHubUser>(&client, &url, token)
        .await
        .with_context(|| format!("fetching user '{username}'"))
}

/// Wrapper for the GitHub search/repositories response.
#[derive(Debug, Deserialize)]
struct SearchReposResponse {
    items: Vec<GitHubRepo>,
}

/// Fetch a user's top starred repositories via the Search API.
///
/// Uses `/search/repositories` because `/users/{u}/repos` does not support `sort=stars`.
pub async fn get_user_starred_repos(
    username: &str,
    limit: usize,
    token: Option<&str>,
) -> anyhow::Result<Vec<GitHubRepo>> {
    let client = build_client()?;
    let per_page = limit.min(100);
    let url = format!(
        "{API_BASE}/search/repositories?q=user:{username}+fork:false&sort=stars&order=desc&per_page={per_page}"
    );
    get_json::<SearchReposResponse>(&client, &url, token)
        .await
        .map(|r| r.items)
        .with_context(|| format!("fetching starred repos for '{username}'"))
}

/// Fetch a user's own repositories sorted by `sort` (`pushed` or `updated`).
///
/// `limit` is capped at 100 (GitHub's maximum per-page).
/// Only returns repos the user owns directly (`type=owner`).
pub async fn get_user_repos(
    username: &str,
    sort: &str,
    limit: usize,
    token: Option<&str>,
) -> anyhow::Result<Vec<GitHubRepo>> {
    let client = build_client()?;
    let per_page = limit.min(100);
    let url = format!(
        "{API_BASE}/users/{username}/repos?type=owner&sort={sort}&direction=desc&per_page={per_page}"
    );
    get_json::<Vec<GitHubRepo>>(&client, &url, token)
        .await
        .with_context(|| format!("fetching repos for '{username}' (sort={sort})"))
}

/// Fetch a user's recent public events (max 100, GitHub returns up to 90 days).
pub async fn get_user_events(
    username: &str,
    limit: usize,
    token: Option<&str>,
) -> anyhow::Result<Vec<GitHubEvent>> {
    let client = build_client()?;
    let per_page = limit.min(100);
    let url = format!("{API_BASE}/users/{username}/events/public?per_page={per_page}");
    get_json::<Vec<GitHubEvent>>(&client, &url, token)
        .await
        .with_context(|| format!("fetching events for '{username}'"))
}

/// Response envelope for the commits search endpoint.
#[derive(Deserialize)]
struct CommitSearchResponse {
    items: Vec<CommitSearchItem>,
}

#[derive(Deserialize)]
struct CommitSearchItem {
    sha: String,
```

```rust
    repository: CommitSearchRepo,
    commit: CommitSearchMeta,
}

#[derive(Deserialize)]
struct CommitSearchRepo {
    full_name: String,
}

#[derive(Deserialize)]
struct CommitSearchMeta {
    message: String,
}

/// Search for the `limit` most recent public commits authored by `username` across all repos.
///
/// Uses `GET /search/commits?q=author:{username}` (stable since GitHub API v3 2022+).
/// Returns `(owner/repo, sha, first-line-of-message)` tuples, newest first.
/// Returns an empty Vec on error so the caller can degrade gracefully.
pub async fn search_user_commits(
    username: &str,
    limit: usize,
    token: Option<&str>,
) -> anyhow::Result<Vec<(String, String, String)>> {
    let client = build_client()?;
    let per_page = limit.min(100);
    let url = format!(
        "{API_BASE}/search/commits?q=author:{username}&sort=committer-date&order=desc&per_page={per_page}"
    );
    get_json::<CommitSearchResponse>(&client, &url, token)
        .await
        .map(|r| {
            r.items
                .into_iter()
                .map(|item| {
                    let msg = item
                        .commit
                        .message
                        .lines()
                        .next()
                        .unwrap_or(&item.commit.message)
                        .to_string();
                    (item.repository.full_name, item.sha, msg)
                })
                .collect()
        })
        .with_context(|| format!("searching commits by '{username}'"))
}

/// Fetch a single commit with its file patches.
pub async fn get_commit_detail(
    owner_repo: &str,
    sha: &str,
    token: Option<&str>,
) -> anyhow::Result<CommitDetail> {
    let client = build_client()?;
    let url = format!("{API_BASE}/repos/{owner_repo}/commits/{sha}");
    get_json::<CommitDetail>(&client, &url, token)
        .await
        .with_context(|| format!("fetching commit {sha} in {owner_repo}"))
}

#[cfg(test)]
mod tests {
    use super::*;
    use httpmock::prelude::*;

    #[test]
    fn auth_header_some() {
        assert_eq!(auth_header(Some("tok")), Some("Bearer tok".to_string()));
    }

    #[test]
    fn auth_header_none() {
        assert_eq!(auth_header(None), None);
    }

    #[tokio::test]
```

```rust
async fn parses_user_response() -> anyhow::Result<()> {
    let server = MockServer::start();
    server.mock(|when, then| {
        when.method(GET).path("/users/alice");
        then.status(200).json_body(serde_json::json!({
            "login": "alice", "name": "Alice", "bio": null, "location": null,
            "company": null, "blog": null, "email": null, "public_repos": 10,
            "followers": 42, "following": 5, "created_at": "2020-01-01T00:00:00Z",
            "html_url": "https://github.com/alice"
        }));
    });

    let client = build_client()?;
    let user: GitHubUser =
        get_json(&client, &format!("{}/users/alice", server.base_url()), None).await?;
    assert_eq!(user.login, "alice");
    assert_eq!(user.public_repos, 10);
    assert_eq!(user.followers, 42);
    Ok(())
}

#[tokio::test]
async fn parses_repo_list_response() -> anyhow::Result<()> {
    let server = MockServer::start();
    server.mock(|when, then| {
        when.method(GET).path("/users/alice/repos");
        then.status(200).json_body(serde_json::json!([{
            "name": "myrepo", "full_name": "alice/myrepo",
            "html_url": "https://github.com/alice/myrepo", "description": null,
            "language": "Rust", "stargazers_count": 7, "forks_count": 1,
            "pushed_at": "2024-03-01T00:00:00Z", "updated_at": "2024-03-01T00:00:00Z",
            "fork": false
        }]));
    });

    let client = build_client()?;
    let repos: Vec<GitHubRepo> = get_json(
        &client,
        &format!("{}/users/alice/repos", server.base_url()),
        None,
    )
    .await?;
    assert_eq!(repos.len(), 1);
    assert_eq!(repos[0].name, "myrepo");
    assert_eq!(repos[0].stargazers_count, 7);
    Ok(())
}

#[tokio::test]
async fn parses_event_list_response() -> anyhow::Result<()> {
    let server = MockServer::start();
    server.mock(|when, then| {
        when.method(GET).path("/users/alice/events/public");
        then.status(200).json_body(serde_json::json!([{
            "type": "PushEvent",
            "repo": { "name": "alice/myrepo" },
            "payload": { "ref": "refs/heads/main", "commits": [] },
            "created_at": "2024-03-01T12:00:00Z"
        }]));
    });

    let client = build_client()?;
    let events: Vec<GitHubEvent> = get_json(
        &client,
        &format!("{}/users/alice/events/public", server.base_url()),
        None,
    )
    .await?;
    assert_eq!(events.len(), 1);
    assert_eq!(events[0].kind, "PushEvent");
    assert_eq!(events[0].repo.name, "alice/myrepo");
    Ok(())
}

#[tokio::test]
async fn parses_commit_detail_response() -> anyhow::Result<()> {
    let server = MockServer::start();
    let sha = "abc1234abc1234abc1234abc1234abc1234abc1234";
```

```rust
        server.mock(|when, then| {
            when.method(GET)
                .path(format!("/repos/alice/myrepo/commits/{sha}"));
            then.status(200).json_body(serde_json::json!({
                "sha": sha,
                "html_url": "https://github.com/alice/myrepo/commit/abc1234",
                "commit": {
                    "message": "fix: handle edge case",
                    "author": { "name": "Alice", "date": "2024-03-01T12:00:00Z" }
                },
                "files": [{
                    "filename": "src/lib.rs", "status": "modified",
                    "additions": 5, "deletions": 2, "patch": "+added line\n-removed line"
                }]
            }));
        });

        let client = build_client()?;
        let detail: CommitDetail = get_json(
            &client,
            &format!("{}/repos/alice/myrepo/commits/{sha}", server.base_url()),
            None,
        )
        .await?;
        assert_eq!(detail.sha, sha);
        assert_eq!(detail.commit.message, "fix: handle edge case");
        assert_eq!(detail.files[0].additions, 5);
        Ok(())
    }

    #[tokio::test]
    async fn rate_limit_error_is_surfaced() {
        let server = MockServer::start();
        server.mock(|when, then| {
            when.method(GET).path("/users/alice");
            then.status(403);
        });

        let client = build_client().unwrap();
        let err =
            get_json::<GitHubUser>(&client, &format!("{}/users/alice", server.base_url()), None)
                .await
                .unwrap_err();
        assert!(err.to_string().contains("rate limit"), "got: {err}");
    }
}
```

src/highlight.rs 242 LOC · 7.4 KB · 2026-02-18

```rust
use std::path::Path;

use syntect::easy::HighlightLines;
use syntect::highlighting::{FontStyle, ThemeSet};
use syntect::parsing::SyntaxSet;

use crate::types::{HighlightedLine, HighlightedToken, RgbColor};

/// Syntax highlighter backed by the bundled syntect theme and syntax sets.
pub struct Highlighter {
    syntax_set: SyntaxSet,
    theme: syntect::highlighting::Theme,
}

impl Highlighter {
    /// Creates a new `Highlighter` using the named syntect theme.
    ///
    /// Theme names are the keys returned by [`list_themes`]. Pass `"InspiredGitHub"` for
    /// the default light theme.
    ///
    /// # Errors
    ///
    /// Returns an error if `theme_name` is not found in the bundled theme set.
    ///
    /// # Examples
    ///
    /// ```
    /// use gitprint::highlight::Highlighter;
    ///
    /// let hl = Highlighter::new("InspiredGitHub").unwrap();
    ///
    /// let err = Highlighter::new("no-such-theme").err().unwrap();
    /// assert!(err.to_string().contains("no-such-theme"));
    /// ```
    pub fn new(theme_name: &str) -> anyhow::Result<Self> {
        let syntax_set = SyntaxSet::load_defaults_newlines();
        let theme_set = ThemeSet::load_defaults();

        let theme = theme_set.themes.get(theme_name).cloned().ok_or_else(|| {
            anyhow::anyhow!(
                "theme not found: {theme_name} (use --list-themes to see available themes)"
            )
        })?;

        Ok(Self { syntax_set, theme })
    }

    /// Returns a lazy iterator that yields one [`HighlightedLine`] at a time.
    ///
    /// Syntax is detected from the file extension of `path`; unknown extensions fall
    /// back to plain text. Line numbers start at 1.
    ///
    /// # Examples
    ///
    /// ```
    /// use gitprint::highlight::Highlighter;
    /// use std::path::Path;
    ///
    /// let hl = Highlighter::new("InspiredGitHub").unwrap();
    /// let lines: Vec<_> = hl.highlight_lines("fn main() {}", Path::new("main.rs")).collect();
    ///
    /// assert_eq!(lines.len(), 1);
    /// assert_eq!(lines[0].line_number, 1);
    /// assert!(!lines[0].tokens.is_empty());
    /// ```
    pub fn highlight_lines<'a>(
        &'a self,
        content: &'a str,
        path: &Path,
    ) -> impl Iterator<Item = HighlightedLine> + 'a {
        let syntax = self
            .syntax_set
            .find_syntax_for_file(path)
            .ok()
            .flatten()
            .unwrap_or_else(|| self.syntax_set.find_syntax_plain_text());

```

```
        let mut h = HighlightLines::new(syntax, &self.theme);
        let mut lines = content.lines().enumerate();

        std::iter::from_fn(move || {
            let (i, line_text) = lines.next()?;

            let tokens = h
                .highlight_line(line_text, &self.syntax_set)
                .unwrap_or_default()
                .into_iter()
                .map(|(style, text)| HighlightedToken {
                    text: text.to_string(),
                    color: RgbColor {
                        r: style.foreground.r,
                        g: style.foreground.g,
                        b: style.foreground.b,
                    },
                    bold: style.font_style.contains(FontStyle::BOLD),
                    italic: style.font_style.contains(FontStyle::ITALIC),
                })
                .collect();

            Some(HighlightedLine {
                line_number: i + 1,
                tokens,
            })
        })
    }
}

/// Returns all available theme names in sorted order.
///
/// # Examples
///
/// ```
/// use gitprint::highlight::list_themes;
///
/// let themes = list_themes();
/// assert!(themes.contains(&"InspiredGitHub".to_string()));
/// assert!(themes.windows(2).all(|w| w[0] <= w[1])); // sorted
/// ```
pub fn list_themes() -> Vec<String> {
    let mut themes: Vec<_> = ThemeSet::load_defaults().themes.into_keys().collect();
    themes.sort();
    themes
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn new_with_valid_theme() {
        assert!(Highlighter::new("InspiredGitHub").is_ok());
    }

    #[test]
    fn new_with_another_valid_theme() {
        assert!(Highlighter::new("base16-ocean.dark").is_ok());
    }

    #[test]
    fn new_with_invalid_theme() {
        let result = Highlighter::new("NonExistentTheme");
        assert!(result.is_err());
        assert!(
            result
                .err()
                .unwrap()
                .to_string()
                .contains("NonExistentTheme")
        );
    }

    #[test]
    fn highlight_lines_produces_output() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let lines: Vec<_> = h
```

```rust
            .highlight_lines("fn main() {}", Path::new("test.rs"))
            .collect();
        assert_eq!(lines.len(), 1);
        assert_eq!(lines[0].line_number, 1);
        assert!(!lines[0].tokens.is_empty());
    }

    #[test]
    fn highlight_lines_multiline() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let content = "line1\nline2\nline3";
        let lines: Vec<_> = h.highlight_lines(content, Path::new("test.txt")).collect();
        assert_eq!(lines.len(), 3);
        assert_eq!(lines[0].line_number, 1);
        assert_eq!(lines[1].line_number, 2);
        assert_eq!(lines[2].line_number, 3);
    }

    #[test]
    fn highlight_lines_preserves_text() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let content = "hello world";
        let lines: Vec<_> = h.highlight_lines(content, Path::new("test.txt")).collect();
        let reconstructed: String = lines[0].tokens.iter().map(|t| t.text.as_str()).collect();
        assert_eq!(reconstructed, "hello world");
    }

    #[test]
    fn highlight_lines_plain_text_fallback() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let lines: Vec<_> = h
            .highlight_lines("some content", Path::new("file.xyz"))
            .collect();
        assert_eq!(lines.len(), 1);
        assert!(!lines[0].tokens.is_empty());
    }

    #[test]
    fn highlight_lines_empty_content() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let lines: Vec<_> = h.highlight_lines("", Path::new("empty.rs")).collect();
        assert!(lines.is_empty());
    }

    #[test]
    fn highlight_lines_rust_code_has_colors() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let content = "fn main() {\n    let x = 42;\n}";
        let lines: Vec<_> = h.highlight_lines(content, Path::new("main.rs")).collect();
        assert_eq!(lines.len(), 3);
        assert!(!lines[0].tokens.is_empty());
    }

    #[test]
    fn highlight_tokens_have_rgb_colors() {
        let h = Highlighter::new("InspiredGitHub").unwrap();
        let lines: Vec<_> = h.highlight_lines("let x = 1;", Path::new("t.rs")).collect();
        lines[0].tokens.iter().for_each(|token| {
            let _ = (token.color.r, token.color.g, token.color.b);
        });
    }

    #[test]
    fn list_themes_non_empty() {
        assert!(!list_themes().is_empty());
    }

    #[test]
    fn list_themes_contains_known_theme() {
        assert!(list_themes().contains(&"InspiredGitHub".to_string()));
    }

    #[test]
    fn list_themes_is_sorted() {
        let themes = list_themes();
        let mut sorted = themes.clone();
        sorted.sort();
        assert_eq!(themes, sorted);
```

```
    }

    #[test]
    fn list_themes_contains_multiple() {
        let themes = list_themes();
        assert!(themes.len() > 1);
        assert!(themes.contains(&"base16-ocean.dark".to_string()));
    }
}
```

src/lib.rs 525 LOC · 17.8 KB · 2026-03-14

```rust
//! # gitprint
//!
//! Convert git repositories into syntax-highlighted, printer-friendly PDFs.
//!
//! The main entry point is [`run()`], which executes the full pipeline:
//! git repository inspection, file filtering, syntax highlighting, and PDF generation.

#![warn(missing_docs)]

/// Command-line argument parsing via Clap.
pub mod cli;
/// Default glob patterns excluded from PDF output.
pub mod defaults;
/// Glob-based file filtering and binary/minified detection.
pub mod filter;
/// Git operations via subprocess.
pub mod git;
/// GitHub REST API v3 client.
pub mod github;
/// Syntax highlighting via syntect.
pub mod highlight;
/// PDF generation via printpdf.
pub mod pdf;
/// Terminal preview renderer.
pub mod preview;
/// Shared data types.
pub mod types;
/// GitHub user activity report pipeline.
pub mod user_report;

use std::path::{Path, PathBuf};
use std::sync::Arc;

use anyhow::bail;

use crate::types::{Config, HighlightedLine};

/// A processed file ready for PDF rendering.
struct ProcessedFile {
    path: PathBuf,
    lines: Vec<HighlightedLine>,
    line_count: usize,
    /// Pre-formatted size string, computed once to avoid calling format_size twice.
    size_str: String,
    last_modified: String,
}

pub(crate) fn format_size(bytes: u64) -> String {
    if bytes < 1024 {
        format!("{bytes} B")
    } else if bytes < 1024 * 1024 {
        format!("{:.1} KB", bytes as f64 / 1024.0)
    } else {
        format!("{:.1} MB", bytes as f64 / (1024.0 * 1024.0))
    }
}

/// Formats the current UTC time as `YYYY-MM-DD HH:MM:SS UTC`.
///
/// Uses Howard Hinnant's Euclidean Gregorian algorithm — no external crate needed.
pub(crate) fn format_utc_now() -> String {
    let total_secs = std::time::SystemTime::now()
        .duration_since(std::time::UNIX_EPOCH)
        .map(|d| d.as_secs())
        .unwrap_or(0);

    let (h, m, s) = (
        (total_secs / 3600) % 24,
        (total_secs / 60) % 60,
        total_secs % 60,
    );

    let z = (total_secs / 86400) as i64 + 719_468;
    let era = z.div_euclid(146_097);
    let doe = z - era * 146_097;
    let yoe = (doe - doe / 1460 + doe / 36524 - doe / 146096) / 365;
    let y = yoe + era * 400;
```

```rust
    let doy = doe - (365 * yoe + yoe / 4 - yoe / 100);
    let mp = (5 * doy + 2) / 153;
    let d = doy - (153 * mp + 2) / 5 + 1;
    let mo = if mp < 10 { mp + 3 } else { mp - 9 };
    let y = if mo <= 2 { y + 1 } else { y };

    format!("{y:04}-{mo:02}-{d:02} {h:02}:{m:02}:{s:02} UTC")
}

fn format_elapsed(elapsed: std::time::Duration) -> String {
    if elapsed.as_millis() < 1000 {
        format!("{}ms", elapsed.as_millis())
    } else {
        format!("{:.1}s", elapsed.as_secs_f64())
    }
}

/// Runs the full gitprint pipeline and writes a PDF to `config.output_path`.
///
/// Accepts a single file, a git repository (optionally scoped to a subdirectory),
/// or a plain directory. The output always goes to `config.output_path`.
///
/// # Errors
///
/// Returns an error if the path does not exist, git operations fail, the theme is
/// invalid, or writing the PDF fails.
///
/// # Examples
///
/// ```ignore
/// use gitprint::types::{Config, PaperSize};
/// use std::path::PathBuf;
///
/// let config = Config {
///     repo_path: PathBuf::from("."),
///     output_path: PathBuf::from("out.pdf"),
///     // ... other fields
/// #   include_patterns: vec![],
/// #   exclude_patterns: vec![],
/// #   theme: "InspiredGitHub".to_string(),
/// #   font_size: 8.0,
/// #   no_line_numbers: false,
/// #   toc: true,
/// #   file_tree: true,
/// #   branch: None,
/// #   commit: None,
/// #   paper_size: PaperSize::A4,
/// #   landscape: false,
/// };
/// gitprint::run(&config).await.unwrap();
/// ```
///
/// **Concurrency model**:
/// - Single-file mode: highlighter init (CPU, `spawn_blocking`) runs concurrently with
///   file content read and last-modified date fetch (both I/O).
/// - Multi-file mode: git metadata, tracked-file list, date map, and highlighter init
///   all run concurrently via `tokio::join!`; highlighter uses `spawn_blocking` to keep
///   tokio worker threads free for I/O.
/// - File reads use a tokio `JoinSet` (I/O-bound parallelism).
/// - Syntax highlighting uses a tokio `JoinSet` of `spawn_blocking` tasks — one per file
///   — so all files are highlighted concurrently across the blocking thread pool (CPU-bound).
/// - Cover, TOC, and tree PDF renders are sequential (each < 5 ms; not worth the overhead).
pub async fn run(config: &Config) -> anyhow::Result<()> {
    let start = std::time::Instant::now();

    let info = git::verify_repo(&config.repo_path).await?;

    // Single-file mode: no cover page, TOC, or file tree — just render the file.
    if let Some(ref single_file) = info.single_file {
        // Highlighter init (CPU, spawn_blocking) overlaps with two I/O calls.
        let theme = config.theme.clone();
        let (highlighter_res, content_res, last_modified) = tokio::join!(
            tokio::task::spawn_blocking(move || highlight::Highlighter::new(&theme)),
            git::read_file_content(&info.root, single_file, config),
            git::file_last_modified(&info.root, single_file, config, info.is_git),
        );
        let highlighter =
            highlighter_res.map_err(|e| anyhow::anyhow!("highlighter panicked: {e}"))??;
```

```
        let content = content_res?;

        if filter::is_binary(content.as_bytes()) || filter::is_minified(&content) {
            bail!("{}: binary or minified file", single_file.display());
        }
        let line_count = content.lines().count();
        let size_str = format_size(content.len() as u64);
        let lines: Vec<HighlightedLine> =
            highlighter.highlight_lines(&content, single_file).collect();

        let doc_title = config
            .remote_url
            .as_deref()
            .map(git::repo_name_from_url)
            .unwrap_or_else(|| {
                config
                    .repo_path
                    .file_name()
                    .map(|n| n.to_string_lossy().to_string())
                    .unwrap_or_else(|| "gitprint".to_string())
            });
        let mut doc = printpdf::PdfDocument::new(&doc_title);
        let fonts = pdf::fonts::load_fonts(&mut doc)?;
        let mut builder = pdf::create_builder(config, fonts);
        let file_info = format!("{line_count} LOC \u{00B7} {size_str} \u{00B7} {last_modified}");
        let header_url = config.remote_url.as_ref().map(|url| {
            let base = url.trim_end_matches(".git");
            format!("{base}/blob/HEAD/{}", single_file.display())
        });
        pdf::code::render_file(
            &mut builder,
            &single_file.display().to_string(),
            lines.into_iter(),
            line_count,
            !config.no_line_numbers,
            config.font_size as u8,
            &file_info,
            header_url.as_deref(),
        );
        let pages = builder.finish();
        let total_pages = pages.len();
        doc.with_pages(pages);
        pdf::save_pdf(&doc, &config.output_path).await?;

        let elapsed = start.elapsed();
        let pdf_size = tokio::fs::metadata(&config.output_path)
            .await
            .map(|m| m.len())
            .unwrap_or(0);
        eprintln!(
            "{} — 1 file, {} pages, {}, {}",
            config.output_path.display(),
            total_pages,
            format_size(pdf_size),
            format_elapsed(elapsed),
        );
        return Ok(());
    }

    let repo_path = info.root;
    let is_git = info.is_git;
    let scope = info.scope;

    // Parallel: git metadata + tracked file list + date map + highlighter init
    // + fs owner/group + repo disk size (for local paths).
    // Highlighter::new is CPU-bound (syntect deserialization); spawn_blocking keeps
    // tokio worker threads free for the concurrent I/O-bound git calls.
    let theme = config.theme.clone();
    let fs_path = config.repo_path.clone();
    let fs_path2 = repo_path.clone();
    let is_remote = config.remote_url.is_some();
    let generated_at = format_utc_now();
    let repo_path_for_git_size = repo_path.clone();
    let config_for_git_size = config.clone();
    let (
        metadata_res,
        all_paths_res,
        date_map_res,
```

```
        highlighter_res,
        fs_owner_group,
        git_repo_size,
        fs_size,
    ) = tokio::join!(
        git::get_metadata(&repo_path, config, is_git, scope.as_deref()),
        git::list_tracked_files(&repo_path, config, is_git, scope.as_deref()),
        git::file_last_modified_dates(&repo_path, config, is_git, scope.as_deref()),
        tokio::task::spawn_blocking(move || highlight::Highlighter::new(&theme)),
        async move {
            if is_remote {
                (None, None)
            } else {
                git::fs_owner_group(&fs_path).await
            }
        },
        async move {
            if is_git {
                git::git_tracked_size(&repo_path_for_git_size, &config_for_git_size).await
            } else {
                String::new()
            }
        },
        async move {
            if is_remote {
                String::new()
            } else {
                git::fs_dir_size(&fs_path2).await
            }
        },
    );

    let mut metadata = metadata_res?;
    if let Some(ref url) = config.remote_url {
        metadata.name = git::repo_name_from_url(url);
    }
    metadata.fs_owner = fs_owner_group.0;
    metadata.fs_group = fs_owner_group.1;
    metadata.generated_at = generated_at;
    metadata.repo_size = git_repo_size;
    metadata.fs_size = fs_size;
    if !is_remote {
        metadata.repo_absolute_path = Some(repo_path.clone());
    }
    let highlighter =
        Arc::new(highlighter_res.map_err(|e| anyhow::anyhow!("highlighter panicked: {e}"))??);
    let date_map = Arc::new(date_map_res?);

    let file_filter = filter::FileFilter::new(&config.include_patterns, &config.exclude_patterns)?;
    let mut paths: Vec<_> = file_filter.filter_paths(all_paths_res?).collect();
    paths.sort_unstable();

    // Phase 1 — I/O: read all file contents concurrently with tokio.
    let mut read_set: tokio::task::JoinSet<Option<(PathBuf, String, String)>> =
        tokio::task::JoinSet::new();
    paths.into_iter().for_each(|path| {
        let repo = repo_path.clone();
        let cfg = config.clone();
        let dates = Arc::clone(&date_map);
        read_set.spawn(async move {
            let content = read_text_file(&repo, &path, &cfg).await?;
            let last_modified = dates.get(&path).cloned().unwrap_or_default();
            Some((path, content, last_modified))
        });
    });
    let raw_files: Vec<(PathBuf, String, String)> =
        read_set.join_all().await.into_iter().flatten().collect();

    // Phase 2 — CPU: highlight each file in a dedicated blocking task so all files
    // are processed concurrently across tokio's blocking thread pool.
    let mut highlight_set: tokio::task::JoinSet<ProcessedFile> = tokio::task::JoinSet::new();
    raw_files
        .into_iter()
        .for_each(|(path, content, last_modified)| {
            let hl = Arc::clone(&highlighter);
            highlight_set.spawn_blocking(move || {
                let line_count = content.lines().count();
                let size_str = format_size(content.len() as u64);
```

```
                let lines: Vec<HighlightedLine> = hl.highlight_lines(&content, &path).collect();
                ProcessedFile {
                    path,
                    lines,
                    line_count,
                    size_str,
                    last_modified,
                }
            });
        });
    let mut files: Vec<ProcessedFile> = highlight_set.join_all().await;

    files.sort_unstable_by(|a, b| a.path.cmp(&b.path));

    metadata.file_count = files.len();
    metadata.total_lines = files.iter().map(|f| f.line_count).sum();

    // Build PDF document and load fonts once.
    let mut doc = printpdf::PdfDocument::new(&metadata.name);
    let fonts = pdf::fonts::load_fonts(&mut doc)?;

    // Collect paths and build dummy TOC entries before the parallel render phase.
    let tree_paths: Vec<PathBuf> = files.iter().map(|f| f.path.clone()).collect();

    // Dummy TOC entries (start_page=0) used purely to count how many pages the TOC occupies.
    // Each entry is one line regardless of content, so page count is stable.
    let dummy_toc_entries: Vec<pdf::toc::TocEntry> = files
        .iter()
        .map(|f| pdf::toc::TocEntry {
            path: f.path.clone(),
            line_count: f.line_count,
            size_str: f.size_str.clone(),
            last_modified: f.last_modified.clone(),
            start_page: 0,
        })
        .collect();

    // For cover links: use explicit remote_url from CLI, or fall back to remote detected
    // from git config so links work even when printing a local repo without --remote.
    let effective_remote_url = config
        .remote_url
        .as_deref()
        .or(metadata.detected_remote_url.as_deref());

    let cover_pages = {
        let mut b = pdf::create_builder(config, fonts.clone());
        pdf::cover::render(&mut b, &metadata, effective_remote_url);
        b.finish()
    };
    let toc_count = if config.toc {
        let mut b = pdf::create_builder(config, fonts.clone());
        pdf::toc::render(&mut b, &dummy_toc_entries);
        b.finish().len()
    } else {
        0
    };
    let tree_count = if config.file_tree {
        let mut b = pdf::create_builder(config, fonts.clone());
        pdf::tree::render(&mut b, &tree_paths);
        b.finish().len()
    } else {
        0
    };
    let cover_count = cover_pages.len();

    // Render file content sequentially, tracking each file's starting page.
    let file_base_page = cover_count + toc_count + tree_count + 1;
    let mut content_builder = pdf::create_builder_at_page(config, fonts.clone(), file_base_page);
    let mut toc_entries: Vec<pdf::toc::TocEntry> = Vec::with_capacity(files.len());

    let remote_base = config.remote_url.as_ref().map(|url| {
        let base = url.trim_end_matches(".git");
        let commit = if metadata.commit_hash.is_empty() {
            "HEAD"
        } else {
            &metadata.commit_hash
        };
        format!("{base}/blob/{commit}")
```

```
    });

    files.into_iter().for_each(|file| {
        let start_page = content_builder.current_page();
        let info = format!(
            "{} LOC \u{00B7} {} \u{00B7} {}",
            file.line_count, file.size_str, file.last_modified
        );
        toc_entries.push(pdf::toc::TocEntry {
            path: file.path.clone(),
            line_count: file.line_count,
            size_str: file.size_str,
            last_modified: file.last_modified.clone(),
            start_page,
        });
        let header_url = remote_base
            .as_ref()
            .map(|base| format!("{base}/{}", file.path.display()));
        pdf::code::render_file(
            &mut content_builder,
            &file.path.display().to_string(),
            file.lines.into_iter(),
            file.line_count,
            !config.no_line_numbers,
            config.font_size as u8,
            &info,
            header_url.as_deref(),
        );
    });
    let content_pages = content_builder.finish();

    let toc_pages = if config.toc {
        let mut b = pdf::create_builder_at_page(config, fonts.clone(), cover_count + 1);
        pdf::toc::render(&mut b, &toc_entries);
        b.finish()
    } else {
        vec![]
    };
    let tree_pages = if config.file_tree {
        let mut b = pdf::create_builder_at_page(config, fonts.clone(), cover_count + toc_count + 1);
        pdf::tree::render(&mut b, &tree_paths);
        b.finish()
    } else {
        vec![]
    };

    // Assemble final document: cover → TOC → tree → file content.
    let all_pages: Vec<_> = cover_pages
        .into_iter()
        .chain(toc_pages)
        .chain(tree_pages)
        .chain(content_pages)
        .collect();
    let total_pages = all_pages.len();

    doc.with_pages(all_pages);
    pdf::save_pdf(&doc, &config.output_path).await?;

    let elapsed = start.elapsed();
    let pdf_size = tokio::fs::metadata(&config.output_path)
        .await
        .map(|m| m.len())
        .unwrap_or(0);

    eprintln!(
        "{} — {} files, {} pages, {}, {}",
        config.output_path.display(),
        metadata.file_count,
        total_pages,
        format_size(pdf_size),
        format_elapsed(elapsed),
    );

    Ok(())
}

async fn read_text_file(repo_path: &Path, path: &Path, config: &Config) -> Option<String> {
    git::read_file_content(repo_path, path, config)
```

```
        .await
        .ok()
        .filter(|c| !filter::is_binary(c.as_bytes()))
        .filter(|c| !filter::is_minified(c))
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn format_size_bytes() {
        assert_eq!(format_size(0), "0 B");
        assert_eq!(format_size(512), "512 B");
        assert_eq!(format_size(1023), "1023 B");
    }

    #[test]
    fn format_size_kilobytes() {
        assert_eq!(format_size(1024), "1.0 KB");
        assert_eq!(format_size(1536), "1.5 KB");
    }

    #[test]
    fn format_size_megabytes() {
        assert_eq!(format_size(1024 * 1024), "1.0 MB");
        assert_eq!(format_size(1024 * 1024 * 2), "2.0 MB");
    }

    #[test]
    fn format_elapsed_milliseconds() {
        assert_eq!(format_elapsed(std::time::Duration::from_millis(0)), "0ms");
        assert_eq!(
            format_elapsed(std::time::Duration::from_millis(999)),
            "999ms"
        );
    }

    #[test]
    fn format_elapsed_seconds() {
        assert_eq!(
            format_elapsed(std::time::Duration::from_millis(1500)),
            "1.5s"
        );
        assert_eq!(format_elapsed(std::time::Duration::from_secs(2)), "2.0s");
    }

    #[test]
    fn format_utc_now_has_correct_format() {
        let s = format_utc_now();
        assert!(s.ends_with(" UTC"), "got: {s}");
        assert_eq!(s.len(), 23, "got: {s}"); // "YYYY-MM-DD HH:MM:SS UTC"
        assert_eq!(&s[4..5], "-");
        assert_eq!(&s[7..8], "-");
        assert_eq!(&s[13..14], ":");
        assert_eq!(&s[16..17], ":");
    }
}
```

src/main.rs 383 LOC · 11.8 KB · 2026-03-05

```rust
use std::path::PathBuf;

use clap::Parser;

/// Parse a human- or machine-readable date string into a `YYYY-MM-DD` string.
///
/// Accepted formats:
/// - ISO 8601: `2024-01-15` or `2024-01-15T00:00:00Z`
/// - Keywords: `today`, `yesterday`
/// - Named: `last week` · `last month` · `last year` · `this week` · `this month` · `this year`
/// - Relative: `N days ago` · `N weeks ago` · `N months ago` · `N years ago`
///   (singular and plural both accepted: "1 day ago" or "2 days ago")
fn parse_date_filter(s: &str) -> anyhow::Result<String> {
    let s = s.trim();
    // ISO 8601: starts with four-digit year followed by '-'
    if s.len() >= 10 && s[..4].parse::<u32>().is_ok() && s.as_bytes().get(4) == Some(&b'-') {
        return Ok(s[..10].to_string());
    }
    let lower = s.to_lowercase();
    let days: u64 = if lower == "today" {
        0
    } else if lower == "yesterday" {
        1
    } else if lower == "last week" || lower == "this week" {
        7
    } else if lower == "last month" || lower == "this month" {
        30
    } else if lower == "last year" || lower == "this year" {
        365
    } else if let Some(n) = lower
        .strip_suffix(" days ago")
        .or_else(|| lower.strip_suffix(" day ago"))
    {
        n.trim()
            .parse::<u64>()
            .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
    } else if let Some(n) = lower
        .strip_suffix(" weeks ago")
        .or_else(|| lower.strip_suffix(" week ago"))
    {
        n.trim()
            .parse::<u64>()
            .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
            * 7
    } else if let Some(n) = lower
        .strip_suffix(" months ago")
        .or_else(|| lower.strip_suffix(" month ago"))
    {
        n.trim()
            .parse::<u64>()
            .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
            * 30
    } else if let Some(n) = lower
        .strip_suffix(" years ago")
        .or_else(|| lower.strip_suffix(" year ago"))
    {
        n.trim()
            .parse::<u64>()
            .map_err(|_| anyhow::anyhow!("invalid date: {s:?}"))?
            * 365
    } else {
        anyhow::bail!(
            "unrecognized date: {s:?}\n\
             Accepted:\n\
             · ISO date:   2024-01-15\n\
             · Keywords:   today · yesterday\n\
             · Named:      last week · last month · last year\n\
             · Relative:   30 days ago · 2 weeks ago · 1 month ago · 1 year ago"
        );
    };
    let secs = std::time::SystemTime::now()
        .duration_since(std::time::UNIX_EPOCH)
        .map_err(|e| anyhow::anyhow!(e))?
        .as_secs()
        .saturating_sub(days * 86_400);
    Ok(unix_secs_to_date(secs))
}
```

```rust

/// Convert a Unix timestamp (seconds, UTC) to a `YYYY-MM-DD` string without external crates.
fn unix_secs_to_date(secs: u64) -> String {
    let mut days = secs / 86_400;
    let mut year = 1970u32;
    loop {
        let in_year = if is_leap_year(year) { 366 } else { 365 };
        if days < in_year {
            break;
        }
        days -= in_year;
        year += 1;
    }
    let month_lengths = if is_leap_year(year) {
        [31u64, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    } else {
        [31u64, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    };
    let mut month = 1u32;
    for &ml in &month_lengths {
        if days < ml {
            break;
        }
        days -= ml;
        month += 1;
    }
    let day = days + 1;
    format!("{year:04}-{month:02}-{day:02}")
}

fn is_leap_year(y: u32) -> bool {
    (y % 4 == 0 && y % 100 != 0) || y % 400 == 0
}

#[tokio::main]
async fn main() {
    let args = gitprint::cli::Args::parse();

    if args.list_themes {
        gitprint::highlight::list_themes()
            .iter()
            .for_each(|t| println!("  {t}"));
        return;
    }

    // ── User report mode ─────────────────────────────────────────────────
    if let Some(username) = args.user {
        let output_path = args
            .output
            .unwrap_or_else(|| PathBuf::from(format!("{username}.pdf")));

        let since = match args.since.as_deref().map(parse_date_filter) {
            Some(Err(e)) => {
                eprintln!("error: --since: {e}");
                std::process::exit(1);
            }
            other => other.and_then(Result::ok),
        };
        let until = match args.until.as_deref().map(parse_date_filter) {
            Some(Err(e)) => {
                eprintln!("error: --until: {e}");
                std::process::exit(1);
            }
            other => other.and_then(Result::ok),
        };

        let config = gitprint::types::UserReportConfig {
            output_path,
            paper_size: args.paper_size,
            landscape: args.landscape,
            last_repos: args.last_repos,
            last_commits: args.last_commits,
            no_diffs: args.no_diffs,
            font_size: args.font_size,
            github_token: std::env::var("GITHUB_TOKEN").ok(),
            since,
            until,
            activity: args.activity,
```

```
            events: args.events,
            username,
        };

        let result = if args.preview {
            gitprint::preview::user(&config).await
        } else {
            gitprint::user_report::run(&config).await
        };
        if let Err(e) = result {
            eprintln!("error: {e:#}");
            std::process::exit(1);
        }
        return;
    }

    // ── Repository mode ──────────────────────────────────────────────────
    let path = match args.path {
        Some(p) => p,
        None => {
            eprintln!("error: a path or -u/--user is required");
            std::process::exit(1);
        }
    };

    let is_remote = gitprint::git::is_remote_url(&path);

    // Clone remote URL to a temp dir; hold it alive until after run().
    let temp_dir = if is_remote {
        match gitprint::git::TempCloneDir::for_url(
            &path,
            args.branch.as_deref(),
            args.commit.as_deref(),
        )
        .await
        {
            Ok(t) => {
                if t.path().join(".git").exists() {
                    eprintln!("Reusing cached clone at {}", t.path().display());
                } else {
                    eprintln!("Cloning {path}...");
                    if let Err(e) = gitprint::git::clone_repo(
                        &path,
                        t.path(),
                        args.branch.as_deref(),
                        args.commit.as_deref(),
                    )
                    .await
                    {
                        eprintln!("error: {e}");
                        std::process::exit(1);
                    }
                }
                Some(t)
            }
            Err(e) => {
                eprintln!("error: {e}");
                std::process::exit(1);
            }
        }
    } else {
        None
    };

    let repo_path = temp_dir
        .as_ref()
        .map(|t| t.path().to_path_buf())
        .unwrap_or_else(|| PathBuf::from(&path));

    if is_remote && args.list_tags {
        if let Err(e) = gitprint::git::fetch_tags(&repo_path).await {
            eprintln!("warning: could not fetch tags: {e}");
        }
    }

    if args.list_tags {
        let tags = gitprint::git::list_repo_tags(&repo_path).await;
        if tags.is_empty() {
```

```rust
            eprintln!("No tags found.");
        } else {
            tags.iter().for_each(|t| println!("{t}"));
        }
        return;
    }

    if args.nvim {
        let status = std::process::Command::new("nvim").arg(&repo_path).status();
        match status {
            Ok(s) => std::process::exit(s.code().unwrap_or(0)),
            Err(e) => {
                eprintln!("error: failed to launch nvim: {e}");
                std::process::exit(1);
            }
        }
    }

    let output_path = args.output.unwrap_or_else(|| {
        let name = if is_remote {
            gitprint::git::repo_name_from_url(&path)
        } else {
            PathBuf::from(&path)
                .canonicalize()
                .ok()
                .and_then(|p| p.file_name().map(|n| n.to_string_lossy().to_string()))
                .unwrap_or_else(|| "output".to_string())
        };
        PathBuf::from(format!("{name}.pdf"))
    });

    let config = gitprint::types::Config {
        repo_path,
        output_path,
        include_patterns: args.include,
        exclude_patterns: args.exclude,
        theme: args.theme,
        font_size: args.font_size,
        no_line_numbers: args.no_line_numbers,
        toc: !args.no_toc,
        file_tree: !args.no_file_tree,
        branch: args.branch,
        commit: args.commit,
        paper_size: args.paper_size,
        landscape: args.landscape,
        remote_url: is_remote.then(|| path.clone()),
    };

    let result = if args.preview {
        gitprint::preview::repo(&config).await
    } else {
        gitprint::run(&config).await
    };
    if let Err(e) = result {
        eprintln!("error: {e}");
        std::process::exit(1);
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn parse_iso_date() -> anyhow::Result<()> {
        assert_eq!(parse_date_filter("2024-01-15")?, "2024-01-15");
        // ISO with time component — truncated to date
        assert_eq!(parse_date_filter("2024-06-30T12:00:00Z")?, "2024-06-30");
        Ok(())
    }

    #[test]
    fn parse_relative_dates() -> anyhow::Result<()> {
        // "today" and "yesterday" produce plausible YYYY-MM-DD strings
        let today = parse_date_filter("today")?;
        assert_eq!(today.len(), 10);
        assert!(today.starts_with("20"));

```

```rust
        let yesterday = parse_date_filter("yesterday")?;
        assert!(yesterday <= today);
        Ok(())
    }

    #[test]
    fn parse_n_days_ago() -> anyhow::Result<()> {
        let d = parse_date_filter("30 days ago")?;
        assert_eq!(d.len(), 10);
        let today = parse_date_filter("today")?;
        assert!(d <= today);
        Ok(())
    }

    #[test]
    fn parse_n_weeks_ago() -> anyhow::Result<()> {
        let d = parse_date_filter("2 weeks ago")?;
        assert_eq!(d.len(), 10);
        Ok(())
    }

    #[test]
    fn parse_n_months_ago() -> anyhow::Result<()> {
        let d = parse_date_filter("1 month ago")?;
        assert_eq!(d.len(), 10);
        Ok(())
    }

    #[test]
    fn parse_n_years_ago() -> anyhow::Result<()> {
        let d = parse_date_filter("1 year ago")?;
        assert_eq!(d.len(), 10);
        let d2 = parse_date_filter("2 years ago")?;
        assert_eq!(d2.len(), 10);
        assert!(d2 <= d);
        Ok(())
    }

    #[test]
    fn parse_named_aliases() -> anyhow::Result<()> {
        let today = parse_date_filter("today")?;
        let last_week = parse_date_filter("last week")?;
        let this_week = parse_date_filter("this week")?;
        assert_eq!(last_week, this_week);
        assert!(last_week <= today);

        let last_month = parse_date_filter("last month")?;
        let this_month = parse_date_filter("this month")?;
        assert_eq!(last_month, this_month);
        assert!(last_month <= last_week);

        let last_year = parse_date_filter("last year")?;
        let this_year = parse_date_filter("this year")?;
        assert_eq!(last_year, this_year);
        assert!(last_year <= last_month);
        Ok(())
    }

    #[test]
    fn parse_invalid_date_errors() {
        assert!(parse_date_filter("not a date").is_err());
        assert!(parse_date_filter("abc days ago").is_err());
    }

    #[test]
    fn unix_secs_known_dates() {
        // 2024-01-01 00:00:00 UTC = 1704067200
        assert_eq!(unix_secs_to_date(1_704_067_200), "2024-01-01");
        // 2000-03-01 (leap year 2000, day after Feb 29)
        assert_eq!(unix_secs_to_date(951_868_800), "2000-03-01");
    }
}
```

src/pdf/code.rs

```rust
use printpdf::{Actions, Color, Pt, Rgb};

use super::layout::{PageBuilder, Span};
use crate::types::HighlightedLine;

/// Renders a syntax-highlighted source file into the PDF, with a file header and optional link.
#[allow(clippy::too_many_arguments)]
pub fn render_file(
    builder: &mut PageBuilder,
    file_path: &str,
    lines: impl Iterator<Item = HighlightedLine>,
    total_lines: usize,
    show_line_numbers: bool,
    font_size: u8,
    file_info: &str,
    // If `Some`, the file header becomes a clickable link to this URL (e.g. GitHub blob view).
    header_url: Option<&str>,
) {
    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
    let size = Pt(font_size as f32);
    let gray = Color::Rgb(Rgb::new(0.59, 0.59, 0.59, None));
    let line_number_width = total_lines.max(1).ilog10() as usize + 1;

    // File header: path left-aligned, metadata right-aligned
    builder.write_line_justified(
        &[Span {
            text: file_path.to_string(),
            font_id: bold,
            size: Pt(font_size as f32 + 2.0),
            color: black,
        }],
        &[Span {
            text: file_info.to_string(),
            font_id: regular,
            size: Pt(7.0),
            color: gray.clone(),
        }],
    );
    if let Some(url) = header_url {
        builder.add_link(builder.line_height(), Actions::Uri(url.to_string()));
    }
    builder.vertical_space(4.0);

    lines.for_each(|line| {
        let mut spans: Vec<Span> = Vec::with_capacity(line.tokens.len() + 1);

        if show_line_numbers {
            spans.push(Span {
                text: format!("{:>width$}  ", line.line_number, width = line_number_width),
                font_id: builder.font(false, false).clone(),
                size,
                color: gray.clone(),
            });
        }

        spans.extend(line.tokens.into_iter().map(|token| Span {
            text: token.text,
            font_id: builder.font(token.bold, token.italic).clone(),
            size,
            color: Color::Rgb(Rgb::new(
                token.color.r as f32 / 255.0,
                token.color.g as f32 / 255.0,
                token.color.b as f32 / 255.0,
                None,
            )),
        }));

        builder.write_line(&spans);
    });

    builder.page_break();
}

#[cfg(test)]
mod tests {
```

```rust
    use crate::pdf;
    use crate::types::{Config, HighlightedLine, HighlightedToken, RgbColor};

    fn sample_lines() -> Vec<HighlightedLine> {
        vec![
            HighlightedLine {
                line_number: 1,
                tokens: vec![HighlightedToken {
                    text: "fn main() {}".into(),
                    color: RgbColor { r: 0, g: 0, b: 0 },
                    bold: false,
                    italic: false,
                }],
            },
            HighlightedLine {
                line_number: 2,
                tokens: vec![HighlightedToken {
                    text: "// comment".into(),
                    color: RgbColor {
                        r: 100,
                        g: 100,
                        b: 100,
                    },
                    bold: false,
                    italic: true,
                }],
            },
        ]
    }

    #[test]
    fn render_file_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render_file(
            &mut builder,
            "test.rs",
            sample_lines().into_iter(),
            2,
            true,
            8,
            "2 lines \u{00B7} 24 B \u{00B7} 2025-01-15",
            None,
        );
    }

    #[test]
    fn render_file_empty_iterator() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render_file(
            &mut builder,
            "empty.rs",
            std::iter::empty(),
            0,
            true,
            8,
            "0 lines \u{00B7} 0 B",
            None,
        );
    }

    #[test]
    fn render_file_without_line_numbers() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render_file(
            &mut builder,
            "test.rs",
            sample_lines().into_iter(),
            2,
            false,
```

```
            8,
            "2 lines \u{00B7} 24 B \u{00B7} 2025-01-15",
            None,
        );
    }

    #[test]
    fn render_file_with_header_url() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render_file(
            &mut builder,
            "src/main.rs",
            sample_lines().into_iter(),
            2,
            true,
            8,
            "2 lines \u{00B7} 24 B \u{00B7} 2025-01-15",
            Some("https://github.com/user/repo/blob/abc123/src/main.rs"),
        );
    }

    #[test]
    fn render_file_many_lines() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let lines: Vec<_> = (1..=100)
            .map(|i| HighlightedLine {
                line_number: i,
                tokens: vec![HighlightedToken {
                    text: format!("line {i}"),
                    color: RgbColor { r: 0, g: 0, b: 0 },
                    bold: false,
                    italic: false,
                }],
            })
            .collect();
        super::render_file(
            &mut builder,
            "big.rs",
            lines.into_iter(),
            100,
            true,
            8,
            "100 lines \u{00B7} 1.2 KB \u{00B7} 2025-01-15",
            None,
        );
    }
}
```

src/pdf/cover.rs

```rust
use std::path::Path;

use printpdf::{Actions, Color, Pt, Rgb};

use super::layout::{PageBuilder, Span};
use crate::types::RepoMetadata;

const CRATES_URL: &str = "https://crates.io/crates/gitprint";
/// Label column width in characters (monospace font — spaces give exact alignment).
const LABEL_COL: usize = 12;
/// Approximate character-width-to-font-size ratio for JetBrains Mono.
const CHAR_WIDTH: f32 = 0.6;

// ── Pure URL-building helpers (also tested independently below) ──────────

/// Extracts a GitHub username from a noreply email address.
///
/// Handles both `123456+username@users.noreply.github.com` and
/// `username@users.noreply.github.com` formats.
fn github_username_from_email(email: &str) -> Option<&str> {
    let local = email.strip_suffix("@users.noreply.github.com")?;
    Some(local.split('+').next_back().unwrap_or(local))
}

/// Returns the URL for a specific commit on the remote.
fn commit_link(remote_base: &str, commit_hash: &str) -> String {
    format!("{remote_base}/commit/{commit_hash}")
}

/// Returns a link to the repo tree at the given commit, or the repo root if no commit.
fn repo_tree_link(remote_base: &str, commit_hash: &str) -> String {
    if commit_hash.is_empty() {
        remote_base.to_string()
    } else {
        format!("{remote_base}/tree/{commit_hash}")
    }
}

/// Returns an author profile/search link for the given email on the remote.
///
/// When the email is a GitHub noreply address the username is extracted and a
/// profile URL (`https://github.com/{user}`) is returned. Otherwise a
/// commit-search-by-email URL is used.
fn author_link(remote_base: &str, email: &str) -> String {
    if let Some(username) = github_username_from_email(email) {
        let host = remote_base
            .splitn(4, '/')
            .take(3)
            .collect::<Vec<_>>()
            .join("/");
        format!("{host}/{username}")
    } else {
        format!("{remote_base}/commits?author={email}")
    }
}

/// Returns a `file://` URL for a local filesystem path.
fn file_url(path: &Path) -> String {
    format!("file://{}", path.display())
}

/// Returns a horizontal rule string that fills `width_pt` at the given `font_size`.
fn separator_line(width_pt: f32, font_size: f32) -> String {
    let chars = (width_pt / (font_size * CHAR_WIDTH)).max(1.0) as usize;
    "─".repeat(chars)
}

// ── Renderer ─────────────────────────────────────────────────────────────

/// Renders the repository cover page, including metadata table and footer.
pub fn render(builder: &mut PageBuilder, metadata: &RepoMetadata, remote_url: Option<&str>) {
    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
    let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
    let lh = builder.line_height();

```

```
    const TABLE_SIZE: f32 = 9.0;

    // Use explicit remote_url if provided; otherwise fall back to the one detected
    // from git config so links work for local git repos without --remote.
    let effective_remote = remote_url.or(metadata.detected_remote_url.as_deref());
    let remote_base = effective_remote.map(|u| u.trim_end_matches(".git"));

    // Title links to repo tree at current commit (remote) or to the local path (local).
    let title_url: Option<String> = remote_base
        .map(|base| repo_tree_link(base, &metadata.commit_hash))
        .or_else(|| metadata.repo_absolute_path.as_deref().map(file_url));

    let commit_url = remote_base
        .filter(|_| !metadata.commit_hash.is_empty())
        .map(|base| commit_link(base, &metadata.commit_hash));

    let author_url = remote_base
        .filter(|_| !metadata.commit_author_email.is_empty())
        .map(|base| author_link(base, &metadata.commit_author_email));

    let author_display = if metadata.commit_author_email.is_empty() {
        metadata.commit_author.clone()
    } else {
        format!(
            "{} <{}>",
            metadata.commit_author, metadata.commit_author_email
        )
    };

    // ── Title ──────────────────────────────────────────────────────────
    builder.vertical_space(120.0);
    builder.write_centered(&metadata.name, &bold, Pt(28.0), black.clone());
    if let Some(url) = title_url {
        builder.add_link(28.0 + 4.0, Actions::Uri(url));
    }
    builder.vertical_space(32.0);

    // ── Metadata table ─────────────────────────────────────────────────
    builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.72, 0.72, 0.72, None)), 0.5);
    builder.vertical_space(8.0);

    // Rows: (label, value, optional URL). Message links to the same commit as Commit.
    [
        ("Branch", metadata.branch.as_str(), None::<String>),
        (
            "Commit",
            metadata.commit_hash_short.as_str(),
            commit_url.clone(),
        ),
        ("Author", author_display.as_str(), author_url),
        ("Date", metadata.commit_date.as_str(), None),
        (
            "Message",
            metadata.commit_message.as_str(),
            commit_url.clone(),
        ),
        ("Files", &metadata.file_count.to_string(), None),
        ("Lines", &metadata.total_lines.to_string(), None),
        ("Repo Size", metadata.repo_size.as_str(), None),
        ("FS Size", metadata.fs_size.as_str(), None),
        ("FS Owner", metadata.fs_owner.as_deref().unwrap_or(""), None),
        ("FS Group", metadata.fs_group.as_deref().unwrap_or(""), None),
        ("Generated", metadata.generated_at.as_str(), None),
    ]
    .into_iter()
    .filter(|(_, value, _)| !value.is_empty())
    .for_each(|(label, value, url)| {
        builder.write_line(&[
            Span {
                text: format!("{label:<LABEL_COL$}"),
                font_id: bold.clone(),
                size: Pt(TABLE_SIZE),
                color: black.clone(),
            },
            Span {
                text: value.into(),
                font_id: regular.clone(),
                size: Pt(TABLE_SIZE),
```

```
                color: black.clone(),
            },
        ]);
        if let Some(u) = url {
            builder.add_link(lh, Actions::Uri(u));
        }
    });

    builder.vertical_space(4.0);
    builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.72, 0.72, 0.72, None)), 0.5);

    // ── Footer (pushed to the bottom of the page) ──────────────────────
    let version = env!("CARGO_PKG_VERSION");
    let footer_text =
        format!("Generated with gitprint v{version} ({CRATES_URL}), a Izel Nakri production");
    let footer_size = Pt(7.0);
    // footer area = separator line (lh) + 4pt gap + footer text (size + 4)
    let footer_area = lh + 4.0 + footer_size.0 + 4.0;
    builder.vertical_space((builder.remaining_pt() - footer_area).max(0.0));

    builder.write_line(&[Span {
        text: separator_line(builder.usable_width_pt(), footer_size.0),
        font_id: regular.clone(),
        size: footer_size,
        color: gray.clone(),
    }]);
    builder.vertical_space(4.0);
    builder.write_centered(&footer_text, &regular, footer_size, gray);
    builder.add_link(footer_size.0 + 4.0, Actions::Uri(CRATES_URL.to_string()));

    builder.page_break();
}

#[cfg(test)]
mod tests {
    use std::path::PathBuf;

    use crate::pdf;
    use crate::types::{Config, RepoMetadata};

    fn test_metadata() -> RepoMetadata {
        RepoMetadata {
            name: "test-repo".into(),
            branch: "main".into(),
            commit_hash: "abc1234567890abcdef1234567890abcdef123456".into(),
            commit_hash_short: "abc1234".into(),
            commit_date: "2024-01-01 12:00:00 +0000".into(),
            commit_message: "initial commit".into(),
            commit_author: "Alice Dev".into(),
            commit_author_email: "alice@example.com".into(),
            file_count: 5,
            total_lines: 100,
            fs_owner: Some("alice".into()),
            fs_group: Some("staff".into()),
            generated_at: "2024-01-15 10:00:00 UTC".into(),
            repo_size: "1.2 MB".into(),
            fs_size: "1.5 MB".into(),
            detected_remote_url: None,
            repo_absolute_path: None,
        }
    }

    // ── URL-building helpers ──────────────────────────────────────────

    #[test]
    fn commit_link_https() {
        assert_eq!(
            super::commit_link("https://github.com/user/repo", "abc123"),
            "https://github.com/user/repo/commit/abc123"
        );
    }

    #[test]
    fn repo_tree_link_with_commit() {
        assert_eq!(
            super::repo_tree_link("https://github.com/user/repo", "abc123"),
            "https://github.com/user/repo/tree/abc123"
        );
```

```rust
    }

    #[test]
    fn repo_tree_link_without_commit_returns_base() {
        assert_eq!(
            super::repo_tree_link("https://github.com/user/repo", ""),
            "https://github.com/user/repo"
        );
    }

    #[test]
    fn author_link_noreply_with_numeric_prefix() {
        assert_eq!(
            super::author_link(
                "https://github.com/user/repo",
                "123456+alice@users.noreply.github.com"
            ),
            "https://github.com/alice"
        );
    }

    #[test]
    fn author_link_noreply_without_numeric_prefix() {
        assert_eq!(
            super::author_link(
                "https://github.com/user/repo",
                "alice@users.noreply.github.com"
            ),
            "https://github.com/alice"
        );
    }

    #[test]
    fn author_link_regular_email_falls_back_to_search() {
        assert_eq!(
            super::author_link("https://github.com/user/repo", "alice@example.com"),
            "https://github.com/user/repo/commits?author=alice@example.com"
        );
    }

    #[test]
    fn file_url_absolute_path() {
        assert_eq!(
            super::file_url(std::path::Path::new("/home/user/project")),
            "file:///home/user/project"
        );
    }

    #[test]
    fn github_username_from_noreply_email() {
        assert_eq!(
            super::github_username_from_email("123456+alice@users.noreply.github.com"),
            Some("alice")
        );
        assert_eq!(
            super::github_username_from_email("alice@users.noreply.github.com"),
            Some("alice")
        );
        assert_eq!(super::github_username_from_email("alice@example.com"), None);
    }

    #[test]
    fn separator_line_fills_width() {
        // At 7.5pt with 0.6 ratio, each char ≈ 4.5pt wide.
        let chars = super::separator_line(45.0, 7.5).chars().count();
        assert_eq!(chars, 10);
    }

    // ── render() smoke tests ──────────────────────────────────────────

    #[test]
    fn render_cover_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(&mut builder, &test_metadata(), None);
        assert!(!builder.finish().is_empty());
```

```
    }

    #[test]
    fn render_cover_with_remote_url() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(
            &mut builder,
            &test_metadata(),
            Some("https://github.com/user/repo"),
        );
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_cover_with_detected_remote_url() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut meta = test_metadata();
        meta.detected_remote_url = Some("https://github.com/user/local-repo".into());
        super::render(&mut builder, &meta, None);
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_cover_with_local_path_file_url() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut meta = test_metadata();
        meta.repo_absolute_path = Some(PathBuf::from("/home/user/myproject"));
        super::render(&mut builder, &meta, None);
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_cover_remote_takes_precedence_over_local_path() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut meta = test_metadata();
        meta.repo_absolute_path = Some(PathBuf::from("/home/user/myproject"));
        super::render(&mut builder, &meta, Some("https://github.com/user/repo"));
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_cover_with_empty_metadata() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(
            &mut builder,
            &RepoMetadata {
                name: String::new(),
                branch: String::new(),
                commit_hash: String::new(),
                commit_hash_short: String::new(),
                commit_date: String::new(),
                commit_message: String::new(),
                commit_author: String::new(),
                commit_author_email: String::new(),
                file_count: 0,
                total_lines: 0,
                fs_owner: None,
                fs_group: None,
                generated_at: String::new(),
                repo_size: String::new(),
                fs_size: String::new(),
                detected_remote_url: None,
                repo_absolute_path: None,
```

```
            },
            None,
        );
    }

    #[test]
    fn render_cover_with_commit_message_is_linked() {
        // Smoke test: cover with remote must not panic with commit_url on message row.
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(
            &mut builder,
            &test_metadata(),
            Some("https://github.com/user/repo.git"),
        );
        assert!(!builder.finish().is_empty());
    }
}
```

src/pdf/diff.rs

```rust
use printpdf::{Actions, Color, Pt, Rgb};

use super::layout::{PageBuilder, Span};
use crate::github::CommitDetail;

// ── Color palette ────────────────────────────────────────────────────────
// Green/red chosen to be distinguishable for common colorblindness types:
//   • Protanopes/deuteranopes see the green as teal-cyan and the red as orange-amber,
//     which remain clearly distinct from each other and from context-line gray.
//   • Both convert to clearly different gray values for black-only printing.
//   • Green is darker (HSL ~150°, 100%, 38%) for better legibility at small sizes.
fn neon_green() -> Color {
    Color::Rgb(Rgb::new(0.0, 0.76, 0.38, None)) // #00C261 — dark electric jade
}
fn neon_red() -> Color {
    Color::Rgb(Rgb::new(0.94, 0.20, 0.20, None)) // #F03333 — deep neon red
}
fn hunk_blue() -> Color {
    Color::Rgb(Rgb::new(0.34, 0.60, 0.96, None)) // #5799F5 — electric blue
}

/// Renders a single commit with its per-file diffs into the PDF.
pub fn render_commit(
    builder: &mut PageBuilder,
    detail: &CommitDetail,
    repo: &str,
    branch: Option<&str>,
    font_size: f32,
) {
    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let italic = builder.font(false, true).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
    let gray = Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None));
    let dark_gray = Color::Rgb(Rgb::new(0.28, 0.28, 0.28, None));
    let rule_gray = Color::Rgb(Rgb::new(0.78, 0.78, 0.78, None));

    let sha_short = detail.sha.get(..7).unwrap_or(&detail.sha);
    let author = &detail.commit.author.name;
    let date = detail
        .commit
        .author
        .date
        .get(..10)
        .unwrap_or(&detail.commit.author.date);
    let message_first_line = detail
        .commit
        .message
        .lines()
        .next()
        .unwrap_or(&detail.commit.message);

    let (total_additions, total_deletions) =
        detail.files.iter().fold((0u64, 0u64), |(add, del), f| {
            (add + f.additions, del + f.deletions)
        });

    builder.ensure_space(builder.line_height() * 5.0);

    // ── Thin separator rule before each commit ──────────────────────────
    builder.draw_horizontal_rule(rule_gray.clone(), 0.4);
    builder.vertical_space(7.0);

    // ── Line 1: sha · message — links to the commit page ────────────────
    builder.write_line(&[
        Span {
            text: format!("{sha_short}  "),
            font_id: bold.clone(),
            size: Pt(font_size),
            color: dark_gray.clone(),
        },
        Span {
            text: message_first_line.to_string(),
            font_id: bold.clone(),
            size: Pt(font_size),
            color: black.clone(),
        },
```

```
    ]);
    builder.add_link(builder.line_height(), Actions::Uri(detail.html_url.clone()));

    // ── Line 2: repo (branch) · author · date · ±stats — links to repo/branch —
    let meta_size = Pt(font_size - 1.0);
    let mut meta_spans = vec![Span {
        text: format!("  {repo}  "),
        font_id: regular.clone(),
        size: meta_size,
        color: dark_gray.clone(),
    }];
    if let Some(b) = branch {
        meta_spans.push(Span {
            text: format!("({b})  "),
            font_id: italic.clone(),
            size: meta_size,
            color: gray.clone(),
        });
    }
    meta_spans.extend([
        Span {
            text: format!("{author}  "),
            font_id: regular.clone(),
            size: meta_size,
            color: dark_gray.clone(),
        },
        Span {
            text: format!("{date}  "),
            font_id: regular.clone(),
            size: meta_size,
            color: gray.clone(),
        },
        Span {
            text: format!("+{total_additions}"),
            font_id: bold.clone(),
            size: meta_size,
            color: neon_green(),
        },
        Span {
            text: "  ".to_string(),
            font_id: regular.clone(),
            size: meta_size,
            color: gray.clone(),
        },
        Span {
            text: format!("-{total_deletions}"),
            font_id: bold.clone(),
            size: meta_size,
            color: neon_red(),
        },
    ]);
    builder.write_line(&meta_spans);
    let meta_url = branch
        .map(|b| format!("https://github.com/{repo}/tree/{b}"))
        .unwrap_or_else(|| format!("https://github.com/{repo}"));
    builder.add_link(builder.line_height(), Actions::Uri(meta_url));

    builder.vertical_space(5.0);

    // ── Per-file diffs ──────────────────────────────────────────────────
    detail.files.iter().for_each(|file| {
        builder.ensure_space(builder.line_height() * 3.0);

        // File header: filename + stats, links to the file at this commit on GitHub.
        builder.write_line(&[
            Span {
                text: format!("  {} ", file.filename),
                font_id: bold.clone(),
                size: Pt(font_size - 0.5),
                color: black.clone(),
            },
            Span {
                text: format!("+{}", file.additions),
                font_id: regular.clone(),
                size: Pt(font_size - 0.5),
                color: neon_green(),
            },
            Span {
```

```
                text: " ".to_string(),
                font_id: regular.clone(),
                size: Pt(font_size - 0.5),
                color: gray.clone(),
            },
            Span {
                text: format!("-{}", file.deletions),
                font_id: regular.clone(),
                size: Pt(font_size - 0.5),
                color: neon_red(),
            },
        ]);
        let file_url = format!(
            "https://github.com/{repo}/blob/{}/{}",
            detail.sha, file.filename
        );
        builder.add_link(builder.line_height(), Actions::Uri(file_url));

        match &file.patch {
            None => {
                builder.write_line(&[Span {
                    text: "  [diff too large to display]".to_string(),
                    font_id: regular.clone(),
                    size: Pt(font_size - 1.0),
                    color: gray.clone(),
                }]);
            }
            Some(patch) => {
                patch.lines().for_each(|line| {
                    let (marker, color) = if line.starts_with('+') {
                        ("+", neon_green())
                    } else if line.starts_with('-') {
                        ("-", neon_red())
                    } else if line.starts_with("@@") {
                        ("@", hunk_blue())
                    } else {
                        (" ", dark_gray.clone())
                    };
                    let body = if line.starts_with("@@") {
                        line.to_string()
                    } else {
                        // Strip the diff prefix char; replace with padded marker.
                        format!("{marker} {}", line.get(1..).unwrap_or(line))
                    };
                    builder.write_line(&[Span {
                        text: format!("    {body}"),
                        font_id: regular.clone(),
                        size: Pt(font_size - 1.0),
                        color,
                    }]);
                });
            }
        }

        builder.vertical_space(3.0);
    });

    builder.vertical_space(6.0);
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::github::{CommitAuthor, CommitFile, CommitInfo};
    use crate::pdf;
    use crate::types::Config;

    fn test_detail(with_patch: bool) -> CommitDetail {
        CommitDetail {
            sha: "abc1234567890".to_string(),
            html_url: "https://github.com/alice/repo/commit/abc1234".to_string(),
            commit: CommitInfo {
                message: "fix: correct off-by-one error\n\nDetailed description.".to_string(),
                author: CommitAuthor {
                    name: "Alice".to_string(),
                    date: "2024-03-01T12:00:00Z".to_string(),
                },
            },
```

```
            files: vec![CommitFile {
                filename: "src/lib.rs".to_string(),
                status: "modified".to_string(),
                additions: 2,
                deletions: 1,
                patch: if with_patch {
                    Some(
                        "@@ -10,7 +10,8 @@\n context line\n-old line\n+new line\n+added line"
                            .to_string(),
                    )
                } else {
                    None
                },
            }],
        }
    }

    #[test]
    fn render_commit_with_patch_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render_commit(
            &mut builder,
            &test_detail(true),
            "alice/repo",
            Some("main"),
            8.0,
        );
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_commit_without_patch_shows_placeholder() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render_commit(&mut builder, &test_detail(false), "alice/repo", None, 8.0);
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_commit_no_files() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut detail = test_detail(false);
        detail.files.clear();
        super::render_commit(&mut builder, &detail, "alice/repo", Some("dev"), 8.0);
        assert!(!builder.finish().is_empty());
    }
}
```

src/pdf/fonts.rs 47 LOC · 1.5 KB · 2026-03-14

```rust
use printpdf::{ParsedFont, PdfDocument};

use super::layout::FontSet;

const REGULAR: &[u8] = include_bytes!("../../fonts/JetBrainsMono-Regular.ttf");
const BOLD: &[u8] = include_bytes!("../../fonts/JetBrainsMono-Bold.ttf");
const ITALIC: &[u8] = include_bytes!("../../fonts/JetBrainsMono-Italic.ttf");
const BOLD_ITALIC: &[u8] = include_bytes!("../../fonts/JetBrainsMono-BoldItalic.ttf");

fn parse_font(bytes: &[u8], label: &str) -> anyhow::Result<ParsedFont> {
    ParsedFont::from_bytes(bytes, 0, &mut Vec::new())
        .ok_or_else(|| anyhow::anyhow!("font loading failed: {label}: failed to parse font"))
}

/// Parses and registers all four JetBrains Mono variants into the PDF document.
pub fn load_fonts(doc: &mut PdfDocument) -> anyhow::Result<FontSet> {
    let regular = parse_font(REGULAR, "regular")?;
    let bold = parse_font(BOLD, "bold")?;
    let italic = parse_font(ITALIC, "italic")?;
    let bold_italic = parse_font(BOLD_ITALIC, "bold-italic")?;

    Ok(FontSet {
        regular: doc.add_font(&regular),
        bold: doc.add_font(&bold),
        italic: doc.add_font(&italic),
        bold_italic: doc.add_font(&bold_italic),
    })
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn load_fonts_succeeds() {
        let mut doc = PdfDocument::new("test");
        assert!(load_fonts(&mut doc).is_ok());
    }

    #[test]
    fn embedded_font_bytes_are_substantial() {
        assert!(REGULAR.len() > 100_000);
        assert!(BOLD.len() > 100_000);
        assert!(ITALIC.len() > 100_000);
        assert!(BOLD_ITALIC.len() > 100_000);
    }
}
```

src/pdf/layout.rs

```rust
use printpdf::{
    Actions, BorderArray, Color, ColorArray, FontId, Line, LinePoint, LinkAnnotation, Mm, Op,
    PaintMode, PdfFontHandle, PdfPage, Polygon, PolygonRing, Pt, Rect, Rgb, TextItem, WindingOrder,
    graphics::Point,
};

/// A styled text span within a line.
pub struct Span {
    /// The text content of this span.
    pub text: String,
    /// The font to use for this span.
    pub font_id: FontId,
    /// The font size in points.
    pub size: Pt,
    /// The fill color for the text.
    pub color: Color,
}

/// Font set for the four standard variants.
#[derive(Clone)]
pub struct FontSet {
    /// Regular (upright, normal weight) font handle.
    pub regular: FontId,
    /// Bold (upright, bold weight) font handle.
    pub bold: FontId,
    /// Italic (oblique, normal weight) font handle.
    pub italic: FontId,
    /// Bold-italic font handle.
    pub bold_italic: FontId,
}

/// Builds PDF pages with simple top-to-bottom text layout.
///
/// Coordinates: printpdf uses bottom-left origin. We track `y` from the top
/// of the usable area (top margin) downward. When converting to printpdf
/// coordinates we do: `pdf_y = page_height - margin - y`.
pub struct PageBuilder {
    pages: Vec<PdfPage>,
    current_ops: Vec<Op>,
    y: f32,
    page_width: Mm,
    page_height: Mm,
    margin: Mm,
    line_height: f32,
    page_count: usize,
    pending_break: bool,
    fonts: FontSet,
}

impl PageBuilder {
    /// Creates a new `PageBuilder` with the given page dimensions, margin, line height, and fonts.
    pub fn new(
        page_width: Mm,
        page_height: Mm,
        margin: Mm,
        line_height: f32,
        fonts: FontSet,
        starting_page: usize,
    ) -> Self {
        let mut builder = Self {
            pages: Vec::new(),
            current_ops: Vec::new(),
            y: 0.0,
            page_width,
            page_height,
            margin,
            line_height,
            page_count: starting_page.saturating_sub(1),
            pending_break: false,
            fonts,
        };
        builder.start_new_page();
        builder
    }

    /// The page number currently being written, accounting for a pending deferred break.
    pub fn current_page(&self) -> usize {
```

```
        if self.pending_break {
            self.page_count + 1
        } else {
            self.page_count
        }
    }

    fn usable_height(&self) -> f32 {
        self.page_height.into_pt().0 - 2.0 * self.margin.into_pt().0
    }

    fn remaining(&self) -> f32 {
        self.usable_height() - self.y
    }

    fn pdf_y(&self) -> Pt {
        Pt(self.page_height.into_pt().0 - self.margin.into_pt().0 - 12.0 - self.y)
    }

    fn left_x(&self) -> Pt {
        self.margin.into_pt()
    }

    fn start_new_page(&mut self) {
        if !self.current_ops.is_empty() {
            self.pages.push(PdfPage::new(
                self.page_width,
                self.page_height,
                std::mem::take(&mut self.current_ops),
            ));
        }

        self.page_count += 1;
        self.y = 0.0;

        let header_text = format!("- {} -", self.page_count);
        let header_x = self.page_width.into_pt().0 / 2.0 - (header_text.len() as f32 * 2.5);
        let header_y = self.page_height.into_pt().0 - self.margin.into_pt().0 + 2.0;
        let header_font = self.fonts.regular.clone();

        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point {
                    x: Pt(header_x),
                    y: Pt(header_y),
                },
            },
            Op::SetFillColor {
                col: Color::Rgb(Rgb::new(0.5, 0.5, 0.5, None)),
            },
            Op::SetFont {
                size: Pt(7.0),
                font: PdfFontHandle::External(header_font.clone()),
            },
            Op::ShowText {
                items: vec![TextItem::Text(header_text)],
            },
            Op::EndTextSection,
        ]);
    }

    /// Flush a deferred page break: start the new page now.
    fn flush_break(&mut self) {
        if self.pending_break {
            self.pending_break = false;
            self.start_new_page();
        }
    }

    /// Ensures at least `needed_pt` of vertical space remains on the current page, breaking if needed.
    pub fn ensure_space(&mut self, needed_pt: f32) {
        self.flush_break();
        if self.remaining() < needed_pt {
            self.start_new_page();
        }
    }

```

```rust
    /// Width in points available for text between the two margins.
    pub fn usable_width_pt(&self) -> f32 {
        self.page_width.into_pt().0 - 2.0 * self.margin.into_pt().0
    }

    /// The line height in points used by this builder.
    pub fn line_height(&self) -> f32 {
        self.line_height
    }

    /// Remaining vertical space in points on the current page.
    pub fn remaining_pt(&self) -> f32 {
        self.usable_height() - self.y
    }

    /// Emits an invisible link annotation covering the last `height_pt` of vertical space written.
    ///
    /// Must be called immediately after the text it should cover (e.g. `write_line*` or
    /// `write_centered`). Pass `builder.line_height()` for a single standard line, or
    /// `n as f32 * builder.line_height()` for n lines (used in TOC). For non-standard font
    /// sizes (e.g. the cover title at 28 pt) pass `font_size + leading` directly.
    ///
    /// The ascender shift is clamped to one line height so multi-row spans don't
    /// shift the entire rect up by their full height.
    pub fn add_link(&mut self, height_pt: f32, action: Actions) {
        // In printpdf, text is placed at its baseline. Visual glyphs extend
        // ~0.7× above (ascenders) and ~0.2× below (descenders) a single line.
        // Shift up by 0.8× of one line so the rect covers what users see.
        let ascender_shift = height_pt.min(self.line_height) * 0.8;
        let y_bottom = Pt(
            self.page_height.into_pt().0 - self.margin.into_pt().0 - 12.0 - self.y + ascender_shift,
        );
        let rect = Rect::from_xywh(
            self.left_x(),
            y_bottom,
            Pt(self.usable_width_pt()),
            Pt(height_pt),
        );
        self.current_ops.push(Op::LinkAnnotation {
            link: LinkAnnotation::new(
                rect,
                action,
                Some(BorderArray::Solid([0.0, 0.0, 0.0])),
                Some(ColorArray::Transparent),
                None,
            ),
        });
    }

    /// Mark a section boundary. The new page is created lazily on the next write,
    /// so finish() never produces a trailing empty page.
    pub fn page_break(&mut self) {
        self.pending_break = true;
    }

    /// Writes a line of styled spans left-aligned at the current cursor position.
    pub fn write_line(&mut self, spans: &[Span]) {
        self.ensure_space(self.line_height);

        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point {
                    x: self.left_x(),
                    y: self.pdf_y(),
                },
            },
        ]);

        self.current_ops.extend(spans.iter().flat_map(|span| {
            [
                Op::SetFillColor {
                    col: span.color.clone(),
                },
                Op::SetFont {
                    size: span.size,
                    font: PdfFontHandle::External(span.font_id.clone()),
                },
```

```
                Op::ShowText {
                    items: vec![TextItem::Text(span.text.clone())],
                },
            ]
        }));

        self.current_ops.push(Op::EndTextSection);
        self.y += self.line_height;
    }

    /// Advances the cursor downward by `pt` points without writing any content.
    pub fn vertical_space(&mut self, pt: f32) {
        self.y += pt;
    }

    /// Writes a single string centered horizontally on the current line.
    pub fn write_centered(&mut self, text: &str, font_id: &FontId, size: Pt, color: Color) {
        self.ensure_space(size.0 + 4.0);

        let text_width = text.len() as f32 * size.0 * 0.6;
        let x = (self.page_width.into_pt().0 - text_width) / 2.0;

        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point {
                    x: Pt(x.max(0.0)),
                    y: self.pdf_y(),
                },
            },
            Op::SetFillColor { col: color },
            Op::SetFont {
                size,
                font: PdfFontHandle::External(font_id.clone()),
            },
            Op::ShowText {
                items: vec![TextItem::Text(text.to_string())],
            },
            Op::EndTextSection,
        ]);

        self.y += size.0 + 4.0;
    }

    /// Writes a line of styled spans centered horizontally on the page.
    pub fn write_line_centered(&mut self, spans: &[Span]) {
        self.ensure_space(self.line_height);
        let y = self.pdf_y();

        let total_width: f32 = spans
            .iter()
            .map(|s| s.text.len() as f32 * s.size.0 * 0.6)
            .sum();
        let x = ((self.page_width.into_pt().0 - total_width) / 2.0).max(0.0);

        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point { x: Pt(x), y },
            },
        ]);
        self.current_ops.extend(spans.iter().flat_map(|span| {
            [
                Op::SetFillColor {
                    col: span.color.clone(),
                },
                Op::SetFont {
                    size: span.size,
                    font: PdfFontHandle::External(span.font_id.clone()),
                },
                Op::ShowText {
                    items: vec![TextItem::Text(span.text.clone())],
                },
            ]
        }));
        self.current_ops.push(Op::EndTextSection);
        self.y += self.line_height;
    }
```

```rust

    /// Writes two groups of spans: `left` aligned to the left margin and `right` to the right margin.
    pub fn write_line_justified(&mut self, left: &[Span], right: &[Span]) {
        self.ensure_space(self.line_height);
        let y = self.pdf_y();

        // Left-aligned spans
        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point {
                    x: self.left_x(),
                    y,
                },
            },
        ]);
        self.current_ops.extend(left.iter().flat_map(|span| {
            [
                Op::SetFillColor {
                    col: span.color.clone(),
                },
                Op::SetFont {
                    size: span.size,
                    font: PdfFontHandle::External(span.font_id.clone()),
                },
                Op::ShowText {
                    items: vec![TextItem::Text(span.text.clone())],
                },
            ]
        }));
        self.current_ops.push(Op::EndTextSection);

        // Right-aligned spans
        let right_width: f32 = right
            .iter()
            .map(|s| s.text.len() as f32 * s.size.0 * 0.6)
            .sum();
        let right_x = self.page_width.into_pt().0 - self.margin.into_pt().0 - right_width;

        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point {
                    x: Pt(right_x.max(0.0)),
                    y,
                },
            },
        ]);
        self.current_ops.extend(right.iter().flat_map(|span| {
            [
                Op::SetFillColor {
                    col: span.color.clone(),
                },
                Op::SetFont {
                    size: span.size,
                    font: PdfFontHandle::External(span.font_id.clone()),
                },
                Op::ShowText {
                    items: vec![TextItem::Text(span.text.clone())],
                },
            ]
        }));
        self.current_ops.push(Op::EndTextSection);

        self.y += self.line_height;
    }

    /// Draw a full-width horizontal rule at the current `y` position and advance
    /// `y` by `thickness_pt` so subsequent content clears the rule.
    pub fn draw_horizontal_rule(&mut self, color: Color, thickness_pt: f32) {
        self.flush_break();
        let y = self.pdf_y();
        let left = self.left_x();
        let right = Pt(left.0 + self.usable_width_pt());
        self.current_ops.extend([
            Op::SaveGraphicsState,
            Op::SetOutlineColor { col: color },
            Op::SetOutlineThickness {
```

```
                pt: Pt(thickness_pt),
            },
            Op::DrawLine {
                line: Line {
                    points: vec![
                        LinePoint {
                            p: Point { x: left, y },
                            bezier: false,
                        },
                        LinePoint {
                            p: Point { x: right, y },
                            bezier: false,
                        },
                    ],
                    is_closed: false,
                },
            },
            Op::RestoreGraphicsState,
        ]);
        self.y += thickness_pt;
    }

    /// Draw a filled rectangle.
    ///
    /// - `x_offset_pt`: x position from the left margin.
    /// - `y_below_cursor_pt`: distance below the current cursor to the **bottom** edge of the rect.
    /// - `width_pt`, `height_pt`: dimensions (rect grows upward from the bottom edge).
    ///
    /// Does **not** advance `y` — call `vertical_space` afterward if needed.
    pub fn draw_filled_rect(
        &mut self,
        x_offset_pt: f32,
        y_below_cursor_pt: f32,
        width_pt: f32,
        height_pt: f32,
        color: Color,
    ) {
        self.flush_break();
        let x = self.left_x().0 + x_offset_pt;
        let y_bottom = self.pdf_y().0 - y_below_cursor_pt;
        let lp = |px: f32, py: f32| LinePoint {
            p: Point {
                x: Pt(px),
                y: Pt(py),
            },
            bezier: false,
        };
        let polygon = Polygon {
            rings: vec![PolygonRing {
                points: vec![
                    lp(x, y_bottom),
                    lp(x + width_pt, y_bottom),
                    lp(x + width_pt, y_bottom + height_pt),
                    lp(x, y_bottom + height_pt),
                ],
            }],
            mode: PaintMode::Fill,
            winding_order: WindingOrder::NonZero,
        };
        self.current_ops.extend([
            Op::SaveGraphicsState,
            Op::SetFillColor { col: color },
            Op::DrawPolygon { polygon },
            Op::RestoreGraphicsState,
        ]);
    }

    /// Write text at a specific x offset from the left margin, at the current `y` cursor.
    /// Does **not** advance `y`.
    pub fn write_text_at_x(
        &mut self,
        x_offset_pt: f32,
        text: &str,
        font_id: &FontId,
        size: Pt,
        color: Color,
    ) {
        self.flush_break();
```

```
        let x = Pt(self.left_x().0 + x_offset_pt);
        self.current_ops.extend([
            Op::StartTextSection,
            Op::SetTextCursor {
                pos: Point { x, y: self.pdf_y() },
            },
            Op::SetFillColor { col: color },
            Op::SetFont {
                size,
                font: PdfFontHandle::External(font_id.clone()),
            },
            Op::ShowText {
                items: vec![TextItem::Text(text.to_string())],
            },
            Op::EndTextSection,
        ]);
    }

    /// Returns the appropriate `FontId` for the requested bold/italic combination.
    pub fn font(&self, bold: bool, italic: bool) -> &FontId {
        match (bold, italic) {
            (true, true) => &self.fonts.bold_italic,
            (true, false) => &self.fonts.bold,
            (false, true) => &self.fonts.italic,
            (false, false) => &self.fonts.regular,
        }
    }

    /// Finalizes all pages and returns them; no trailing empty page is produced.
    pub fn finish(mut self) -> Vec<PdfPage> {
        if !self.current_ops.is_empty() {
            self.pages.push(PdfPage::new(
                self.page_width,
                self.page_height,
                self.current_ops,
            ));
        }
        self.pages
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    fn test_font_set() -> (printpdf::PdfDocument, FontSet) {
        let mut doc = printpdf::PdfDocument::new("test");

        let load =
            |bytes: &[u8]| printpdf::ParsedFont::from_bytes(bytes, 0, &mut Vec::new()).unwrap();

        let fonts = FontSet {
            regular: doc.add_font(&load(include_bytes!(
                "../../fonts/JetBrainsMono-Regular.ttf"
            ))),
            bold: doc.add_font(&load(include_bytes!("../../fonts/JetBrainsMono-Bold.ttf"))),
            italic: doc.add_font(&load(include_bytes!(
                "../../fonts/JetBrainsMono-Italic.ttf"
            ))),
            bold_italic: doc.add_font(&load(include_bytes!(
                "../../fonts/JetBrainsMono-BoldItalic.ttf"
            ))),
        };

        (doc, fonts)
    }

    fn black() -> Color {
        Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None))
    }

    #[test]
    fn builder_creates_at_least_one_page() {
        let (_doc, fonts) = test_font_set();
        let pages = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1).finish();
        assert_eq!(pages.len(), 1);
    }

```

```rust
    #[test]
    fn write_line_adds_content() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line(&[Span {
            text: "hello".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        let pages = builder.finish();
        assert_eq!(pages.len(), 1);
        assert!(pages[0].ops.len() > 2);
    }

    #[test]
    fn page_break_creates_new_page() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line(&[Span {
            text: "page 1".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        builder.page_break();
        builder.write_line(&[Span {
            text: "page 2".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        assert_eq!(builder.finish().len(), 2);
    }

    #[test]
    fn trailing_page_break_does_not_add_empty_page() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line(&[Span {
            text: "content".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        builder.page_break();
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn write_centered_works() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_centered("Title", &fonts.regular, Pt(28.0), black());
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn draw_horizontal_rule_does_not_panic() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
        builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.5, 0.5, 0.5, None)), 0.5);
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn many_lines_cause_page_overflow() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        (0..200).for_each(|_| {
            builder.write_line(&[Span {
                text: "line".into(),
                font_id: fonts.regular.clone(),
                size: Pt(8.0),
                color: black(),
            }]);
        });
        assert!(builder.finish().len() > 1);
```

```rust
    }

    #[test]
    fn write_line_centered_does_not_panic() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line_centered(&[Span {
            text: "centered".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn write_line_justified_does_not_panic() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line_justified(
            &[Span {
                text: "left".into(),
                font_id: fonts.regular.clone(),
                size: Pt(8.0),
                color: black(),
            }],
            &[Span {
                text: "right".into(),
                font_id: fonts.bold.clone(),
                size: Pt(8.0),
                color: black(),
            }],
        );
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn draw_filled_rect_does_not_panic() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
        builder.draw_filled_rect(0.0, 20.0, 100.0, 10.0, black());
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn write_text_at_x_does_not_panic() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_text_at_x(50.0, "hello", &fonts.regular, Pt(8.0), black());
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn font_variants_are_distinct() {
        let (_doc, fonts) = test_font_set();
        let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
        // Each combination must return without panic; IDs may or may not be equal.
        let _ = builder.font(false, false);
        let _ = builder.font(true, false);
        let _ = builder.font(false, true);
        let _ = builder.font(true, true);
    }

    #[test]
    fn usable_width_pt_is_positive() {
        let (_doc, fonts) = test_font_set();
        let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
        assert!(builder.usable_width_pt() > 0.0);
    }

    #[test]
    fn line_height_matches_constructor() {
        let (_doc, fonts) = test_font_set();
        let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 12.5, fonts, 1);
        assert_eq!(builder.line_height(), 12.5);
    }

    #[test]
```

```rust
    fn remaining_pt_decreases_after_write() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        let before = builder.remaining_pt();
        builder.write_line(&[Span {
            text: "x".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        assert!(builder.remaining_pt() < before);
    }

    #[test]
    fn current_page_with_pending_break() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line(&[Span {
            text: "x".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        let page_before = builder.current_page();
        builder.page_break();
        // current_page() should report the upcoming page while break is pending.
        assert_eq!(builder.current_page(), page_before + 1);
    }

    #[test]
    fn vertical_space_reduces_remaining() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
        let before = builder.remaining_pt();
        builder.vertical_space(20.0);
        assert!((builder.remaining_pt() - (before - 20.0)).abs() < 0.01);
    }

    #[test]
    fn ensure_space_forces_page_break_when_tight() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 1);
        // Consume almost all space, then request more than what remains.
        let usable = builder.remaining_pt();
        builder.vertical_space(usable - 5.0);
        let page_before = builder.current_page();
        builder.ensure_space(50.0);
        assert!(builder.current_page() > page_before);
    }

    #[test]
    fn add_link_does_not_panic() {
        let (_doc, fonts) = test_font_set();
        let mut builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts.clone(), 1);
        builder.write_line(&[Span {
            text: "linked text".into(),
            font_id: fonts.regular.clone(),
            size: Pt(8.0),
            color: black(),
        }]);
        builder.add_link(
            10.0,
            printpdf::Actions::Uri("https://example.com".to_string()),
        );
        assert_eq!(builder.finish().len(), 1);
    }

    #[test]
    fn starting_page_offset_is_respected() {
        let (_doc, fonts) = test_font_set();
        let builder = PageBuilder::new(Mm(210.0), Mm(297.0), Mm(10.0), 10.0, fonts, 5);
        assert_eq!(builder.current_page(), 5);
    }
}
```

src/pdf/mod.rs 135 LOC · 4.3 KB · 2026-03-14

```rust
/// Syntax-highlighted source code rendering.
pub mod code;
/// Repository cover page rendering.
pub mod cover;
/// Git diff / commit patch rendering.
pub mod diff;
/// Embedded JetBrains Mono font loading.
pub mod fonts;
/// Core page-layout engine (`PageBuilder`).
pub mod layout;
/// Table of contents rendering.
pub mod toc;
/// Directory tree visualization.
pub mod tree;
/// GitHub user activity feed rendering.
pub mod user_activity;
/// User report cover page rendering.
pub mod user_cover;
/// User repository list rendering.
pub mod user_repos;

use std::path::Path;

use printpdf::{Mm, PdfDocument, PdfSaveOptions};

use crate::types::{Config, PaperSize, UserReportConfig};
use layout::{FontSet, PageBuilder};

fn paper_dimensions(config: &Config) -> (Mm, Mm) {
    let (w, h) = match config.paper_size {
        PaperSize::A4 => (Mm(210.0), Mm(297.0)),
        PaperSize::Letter => (Mm(215.9), Mm(279.4)),
        PaperSize::Legal => (Mm(215.9), Mm(355.6)),
    };
    if config.landscape { (h, w) } else { (w, h) }
}

/// Creates a `PageBuilder` starting at page 1 for the given config and font set.
pub fn create_builder(config: &Config, fonts: FontSet) -> PageBuilder {
    create_builder_at_page(config, fonts, 1)
}

/// Creates a `PageBuilder` starting at an arbitrary page number (used to continue page numbering).
pub fn create_builder_at_page(
    config: &Config,
    fonts: FontSet,
    starting_page: usize,
) -> PageBuilder {
    let (w, h) = paper_dimensions(config);
    let line_height = config.font_size as f32 + 2.0;
    PageBuilder::new(w, h, Mm(10.0), line_height, fonts, starting_page)
}

/// Creates a `PageBuilder` for a user report starting at page 1.
pub fn create_user_builder(config: &UserReportConfig, fonts: FontSet) -> PageBuilder {
    create_user_builder_at_page(config, fonts, 1)
}

/// Creates a `PageBuilder` for a user report starting at an arbitrary page number.
pub fn create_user_builder_at_page(
    config: &UserReportConfig,
    fonts: FontSet,
    starting_page: usize,
) -> PageBuilder {
    let (w, h) = match config.paper_size {
        PaperSize::A4 => (Mm(210.0), Mm(297.0)),
        PaperSize::Letter => (Mm(215.9), Mm(279.4)),
        PaperSize::Legal => (Mm(215.9), Mm(355.6)),
    };
    let (w, h) = if config.landscape { (h, w) } else { (w, h) };
    let line_height = config.font_size as f32 + 2.0;
    PageBuilder::new(w, h, Mm(10.0), line_height, fonts, starting_page)
}

/// Serializes a `PdfDocument` to bytes and writes it to `path` asynchronously.
pub async fn save_pdf(doc: &PdfDocument, path: &Path) -> anyhow::Result<()> {
    let mut warnings = Vec::new();
```

```rust
    let bytes = doc.save(&PdfSaveOptions::default(), &mut warnings);
    tokio::fs::write(path, bytes).await.map_err(Into::into)
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::types::Config;

    #[test]
    fn paper_dimensions_a4() {
        let config = Config::test_default();
        let (w, h) = paper_dimensions(&config);
        assert_eq!(w.0, 210.0);
        assert_eq!(h.0, 297.0);
    }

    #[test]
    fn paper_dimensions_letter() {
        let mut config = Config::test_default();
        config.paper_size = PaperSize::Letter;
        let (w, h) = paper_dimensions(&config);
        assert_eq!(w.0, 215.9);
        assert_eq!(h.0, 279.4);
    }

    #[test]
    fn paper_dimensions_landscape() {
        let mut config = Config::test_default();
        config.landscape = true;
        let (w, h) = paper_dimensions(&config);
        assert_eq!(w.0, 297.0);
        assert_eq!(h.0, 210.0);
    }

    #[tokio::test]
    async fn save_pdf_to_tempfile() {
        let mut doc = PdfDocument::new("test");
        let fonts = fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let builder = create_builder(&config, fonts);
        doc.with_pages(builder.finish());

        let dir = tempfile::tempdir().unwrap();
        let path = dir.path().join("test.pdf");
        assert!(save_pdf(&doc, &path).await.is_ok());
        assert!(path.exists());
        assert!(std::fs::metadata(&path).unwrap().len() > 0);
    }

    #[tokio::test]
    async fn save_pdf_invalid_path() {
        let mut doc = PdfDocument::new("test");
        let _ = fonts::load_fonts(&mut doc).unwrap();
        let result = save_pdf(&doc, Path::new("/nonexistent/dir/test.pdf")).await;
        assert!(result.is_err());
    }
}
```

src/pdf/toc.rs

```rust
use std::path::PathBuf;

use printpdf::{Actions, Color, Destination, Pt, Rgb};

use super::layout::{PageBuilder, Span};

/// A single entry in the Table of Contents.
pub struct TocEntry {
    /// Path to the file relative to the repository root.
    pub path: PathBuf,
    /// Number of lines in the file.
    pub line_count: usize,
    /// Human-readable file size (e.g. "4.2 KB").
    pub size_str: String,
    /// Date the file was last modified (YYYY-MM-DD).
    pub last_modified: String,
    /// PDF page number where this file's content begins.
    pub start_page: usize,
}

/// Split `text` into chunks of at most `max_chars` characters each.
fn wrap_text(text: &str, max_chars: usize) -> Vec<String> {
    if max_chars == 0 || text.is_empty() {
        return vec![text.to_string()];
    }
    let mut chunks = Vec::new();
    let mut remaining = text;
    while !remaining.is_empty() {
        let split_at = remaining
            .char_indices()
            .nth(max_chars)
            .map(|(i, _)| i)
            .unwrap_or(remaining.len());
        chunks.push(remaining[..split_at].to_string());
        remaining = &remaining[split_at..];
    }
    chunks
}

/// Renders the table of contents page with clickable internal links for each entry.
pub fn render(builder: &mut PageBuilder, entries: &[TocEntry]) {
    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));

    builder.write_centered("Table of Contents", &bold, Pt(16.0), black);
    builder.vertical_space(10.0);

    // Approximate character width factors (monospace font approximation).
    const PATH_SIZE: f32 = 8.0;
    const META_SIZE: f32 = 7.0;
    const CHAR_WIDTH: f32 = 0.6;
    const GAP_PT: f32 = 8.0;

    entries.iter().for_each(|entry| {
        let meta = format!(
            "p.{}  {} LOC \u{00B7} {} \u{00B7} {}",
            entry.start_page, entry.line_count, entry.size_str, entry.last_modified
        );
        let meta_width = meta.len() as f32 * META_SIZE * CHAR_WIDTH;
        let available_left = builder.usable_width_pt() - meta_width - GAP_PT;
        let max_chars = (available_left / (PATH_SIZE * CHAR_WIDTH)).max(1.0) as usize;

        let path_str = entry.path.display().to_string();
        let chunks = wrap_text(&path_str, max_chars);
        let row_count = chunks.len();

        // First chunk shares the line with meta; remaining chunks are on their own lines.
        builder.write_line_justified(
            &[Span {
                text: chunks[0].clone(),
                font_id: regular.clone(),
                size: Pt(PATH_SIZE),
                color: gray.clone(),
            }],
            &[Span {
```

```
                text: meta,
                font_id: regular.clone(),
                size: Pt(META_SIZE),
                color: gray.clone(),
            }],
        );
        chunks[1..].iter().for_each(|chunk| {
            builder.write_line(&[Span {
                text: chunk.clone(),
                font_id: regular.clone(),
                size: Pt(PATH_SIZE),
                color: gray.clone(),
            }]);
        });

        builder.add_link(
            builder.line_height() * row_count as f32,
            Actions::Goto(Destination::Xyz {
                page: entry.start_page,
                left: None,
                top: None,
                zoom: None,
            }),
        );
    });

    builder.page_break();
}

#[cfg(test)]
mod tests {
    use crate::pdf;
    use crate::types::Config;
    use std::path::PathBuf;

    fn make_entry(path: &str, lines: usize, page: usize) -> super::TocEntry {
        super::TocEntry {
            path: PathBuf::from(path),
            line_count: lines,
            size_str: "1.2 KB".to_string(),
            last_modified: "2024-01-15".to_string(),
            start_page: page,
        }
    }

    #[test]
    fn wrap_text_short() {
        let chunks = super::wrap_text("short", 20);
        assert_eq!(chunks, vec!["short"]);
    }

    #[test]
    fn wrap_text_exact() {
        let chunks = super::wrap_text("1234567890", 10);
        assert_eq!(chunks, vec!["1234567890"]);
    }

    #[test]
    fn wrap_text_overflow() {
        let chunks = super::wrap_text("1234567890ab", 10);
        assert_eq!(chunks, vec!["1234567890", "ab"]);
    }

    #[test]
    fn wrap_text_empty() {
        let chunks = super::wrap_text("", 10);
        assert_eq!(chunks, vec![""]);
    }

    #[test]
    fn render_toc_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let entries = vec![
            make_entry("src/main.rs", 20, 5),
            make_entry("src/lib.rs", 50, 7),
```

```
        ];
        super::render(&mut builder, &entries);
    }

    #[test]
    fn render_toc_empty_files() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(&mut builder, &[]);
    }

    #[test]
    fn render_toc_many_files() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let entries: Vec<_> = (0..100)
            .map(|i| make_entry("src/file.rs", i * 10, i + 5))
            .collect();
        super::render(&mut builder, &entries);
    }

    #[test]
    fn render_toc_long_path_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let entries = vec![make_entry(
            "src/very/deeply/nested/path/that/is/quite/long/file.rs",
            100,
            3,
        )];
        super::render(&mut builder, &entries);
    }
}
```

src/pdf/tree.rs

```rust
use std::collections::BTreeMap;
use std::path::PathBuf;

use printpdf::{Color, Pt, Rgb};

use super::layout::PageBuilder;

/// A recursive directory tree. BTreeMap keeps entries sorted alphabetically.
struct Tree(BTreeMap<String, Tree>);

impl Tree {
    fn new() -> Self {
        Self(BTreeMap::new())
    }

    fn insert(&mut self, parts: &[&str]) {
        if let [first, rest @ ..] = parts {
            self.0
                .entry(first.to_string())
                .or_insert_with(Tree::new)
                .insert(rest);
        }
    }

    fn to_lines(&self, prefix: &str) -> Vec<String> {
        let last_idx = self.0.len().saturating_sub(1);

        self.0
            .iter()
            .enumerate()
            .flat_map(|(i, (name, child))| {
                let is_last = i == last_idx;
                let connector = if is_last {
                    "\u{2514}\u{2500}\u{2500} "
                } else {
                    "\u{251C}\u{2500}\u{2500} "
                };
                let extension = if is_last { "    " } else { "\u{2502}   " };

                std::iter::once(format!("{prefix}{connector}{name}"))
                    .chain(child.to_lines(&format!("{prefix}{extension}")))
            })
            .collect()
    }
}

/// Renders a directory tree page showing all included file paths in box-drawing style.
pub fn render(builder: &mut PageBuilder, paths: &[PathBuf]) {
    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));

    builder.write_centered("File Tree", &bold, Pt(16.0), black.clone());
    builder.vertical_space(10.0);

    let mut root = Tree::new();
    paths.iter().for_each(|p| {
        let parts: Vec<_> = p
            .components()
            .map(|c| c.as_os_str().to_str().unwrap_or("?"))
            .collect();
        root.insert(&parts);
    });

    root.to_lines("").into_iter().for_each(|line| {
        builder.write_line(&[super::layout::Span {
            text: line,
            font_id: regular.clone(),
            size: Pt(7.0),
            color: black.clone(),
        }]);
    });

    builder.page_break();
}

#[cfg(test)]
```

```rust
mod tests {
    use super::*;

    #[test]
    fn single_file() {
        let mut tree = Tree::new();
        tree.insert(&["src", "main.rs"]);
        let lines = tree.to_lines("");
        assert_eq!(lines.len(), 2);
        assert!(lines[0].contains("src"));
        assert!(lines[1].contains("main.rs"));
    }

    #[test]
    fn nested_structure_with_box_drawing() {
        let mut tree = Tree::new();
        tree.insert(&["src", "main.rs"]);
        tree.insert(&["src", "lib.rs"]);
        tree.insert(&["Cargo.toml"]);
        let lines = tree.to_lines("");
        assert!(lines.len() >= 4);
        let joined = lines.join("\n");
        assert!(joined.contains('\u{251C}'));
        assert!(joined.contains('\u{2514}'));
        assert!(joined.contains('\u{2500}'));
    }

    #[test]
    fn empty_tree() {
        assert!(Tree::new().to_lines("").is_empty());
    }

    #[test]
    fn sorted_output() {
        let mut tree = Tree::new();
        tree.insert(&["z.rs"]);
        tree.insert(&["a.rs"]);
        tree.insert(&["m.rs"]);
        let lines = tree.to_lines("");
        assert!(lines[0].contains("a.rs"));
        assert!(lines[1].contains("m.rs"));
        assert!(lines[2].contains("z.rs"));
    }

    #[test]
    fn deep_nesting() {
        let mut tree = Tree::new();
        tree.insert(&["a", "b", "c", "d", "e.txt"]);
        let lines = tree.to_lines("");
        assert_eq!(lines.len(), 5);
    }

    #[test]
    fn multiple_files_same_directory() {
        let mut tree = Tree::new();
        tree.insert(&["src", "a.rs"]);
        tree.insert(&["src", "b.rs"]);
        tree.insert(&["src", "c.rs"]);
        assert_eq!(tree.to_lines("").len(), 4);
    }

    #[test]
    fn render_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = crate::pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = crate::types::Config::test_default();
        let mut builder = crate::pdf::create_builder(&config, fonts);
        render(
            &mut builder,
            &[
                PathBuf::from("src/main.rs"),
                PathBuf::from("src/lib.rs"),
                PathBuf::from("Cargo.toml"),
            ],
        );
    }
}
```

src/pdf/user_activity.rs 688 LOC · 24.6 KB · 2026-03-14

```rust
use printpdf::{Actions, Color, Pt, Rgb};

use super::layout::{PageBuilder, Span};
use crate::github::GitHubEvent;

/// Renders the "Recent Activity" section, grouping events by date with icons and links.
pub fn render(
    builder: &mut PageBuilder,
    events: &[GitHubEvent],
    commit_msgs: &std::collections::HashMap<String, String>,
) {
    if events.is_empty() {
        return;
    }

    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let italic = builder.font(false, true).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
    let gray = Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None));
    let dark_gray = Color::Rgb(Rgb::new(0.25, 0.25, 0.25, None));
    let rule_gray = Color::Rgb(Rgb::new(0.82, 0.82, 0.82, None));

    // ── Section title ──────────────────────────────────────────────
    builder.ensure_space(builder.line_height() * 3.0);
    builder.write_centered("Recent Activity", &bold, Pt(16.0), black.clone());
    builder.vertical_space(10.0);
    builder.draw_horizontal_rule(rule_gray.clone(), 0.5);
    builder.vertical_space(8.0);

    // ── Events grouped by date ─────────────────────────────────────
    let mut last_date = String::new();
    events.iter().for_each(|event| {
        let date = event.created_at.get(..10).unwrap_or(&event.created_at);
        if date != last_date {
            if !last_date.is_empty() {
                // Thin rule between date groups for visual separation.
                builder.vertical_space(4.0);
                builder.draw_horizontal_rule(rule_gray.clone(), 0.3);
                builder.vertical_space(8.0);
            } else {
                builder.vertical_space(2.0);
            }
            builder.ensure_space(builder.line_height() * 2.0);
            builder.write_line(&[Span {
                text: date.to_string(),
                font_id: bold.clone(),
                size: Pt(9.5),
                color: dark_gray.clone(),
            }]);
            last_date = date.to_string();
            builder.vertical_space(2.0);
        }

        let time = event.created_at.get(11..16).unwrap_or("");
        let description = describe_event(event);
        let icon = event_icon(event);

        // Enrich push events that have no commit info in the payload (force push /
        // rebase). Look up this event's HEAD SHA to get its specific commit message.
        let (main, detail) = if event.kind == "PushEvent" && description.detail.is_empty() {
            let sha = event.payload["head"].as_str().unwrap_or("");
            if let Some(msg) = commit_msgs.get(sha) {
                let branch = event.payload["ref"]
                    .as_str()
                    .unwrap_or("")
                    .trim_start_matches("refs/heads/");
                let enriched_main = format!("Pushed to {} ({branch})", event.repo.name);
                (enriched_main, vec![format!("  {msg}")])
            } else {
                (description.main, description.detail)
            }
        } else {
            (description.main, description.detail)
        };

        let url = event_url(event);
```

```
        builder.write_line(&[
            Span {
                text: format!("{icon} "),
                font_id: bold.clone(),
                size: Pt(8.0),
                color: event_icon_color(event),
            },
            Span {
                text: format!("{time}  "),
                font_id: regular.clone(),
                size: Pt(7.5),
                color: gray.clone(),
            },
            Span {
                text: main,
                font_id: regular.clone(),
                size: Pt(8.5),
                color: black.clone(),
            },
        ]);
        if let Some(u) = &url {
            builder.add_link(builder.line_height(), Actions::Uri(u.clone()));
        }

        // Detail lines (commit messages, PR diff stats, etc.) — also link to the event.
        detail.iter().for_each(|detail_line| {
            builder.write_line(&[
                Span {
                    text: "    ".to_string(),
                    font_id: regular.clone(),
                    size: Pt(7.5),
                    color: gray.clone(),
                },
                Span {
                    text: detail_line.clone(),
                    font_id: italic.clone(),
                    size: Pt(7.5),
                    color: gray.clone(),
                },
            ]);
            if let Some(u) = &url {
                builder.add_link(builder.line_height(), Actions::Uri(u.clone()));
            }
        });
    });

    builder.vertical_space(12.0);
    builder.page_break();
}

// ── Event decorators ──────────────────────────────────────────────────────

/// Single-character icon using Geometric Shapes (U+25A0–U+25FF) — all present
/// in JetBrains Mono and rendered reliably across PDF viewers including PDF.js.
fn event_icon(event: &GitHubEvent) -> &'static str {
    match event.kind.as_str() {
        "PushEvent" => "\u{25B6}",         // ▶ black right-pointing triangle
        "PullRequestEvent" => "\u{25B2}",  // ▲ black up-pointing triangle
        "IssuesEvent" => "\u{25CF}",       // ● black circle
        "IssueCommentEvent" => "\u{25E6}", // ◦ white bullet
        "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => "\u{25CB}", // ○ white circle
        "ReleaseEvent" => "\u{25C6}",      // ◆ black diamond
        "ForkEvent" => "\u{25B7}",         // ▷ white right-pointing triangle
        "WatchEvent" => "\u{25C6}",        // ◆ black diamond (star/highlight)
        "CreateEvent" => "\u{25AA}",       // ▪ black small square
        "DeleteEvent" => "\u{25AB}",       // ▫ white small square
        _ => "\u{00B7}",                   // · middle dot
    }
}

fn event_icon_color(event: &GitHubEvent) -> Color {
    match event.kind.as_str() {
        "PushEvent" => Color::Rgb(Rgb::new(0.27, 0.68, 0.96, None)), // blue
        "PullRequestEvent" => Color::Rgb(Rgb::new(0.55, 0.36, 0.90, None)), // purple
        "IssuesEvent" => Color::Rgb(Rgb::new(0.96, 0.55, 0.13, None)), // orange
        "IssueCommentEvent" | "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => {
            Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None)) // gray
        }
```

```
        "ReleaseEvent" => Color::Rgb(Rgb::new(0.13, 0.78, 0.47, None)), // green
        "WatchEvent" => Color::Rgb(Rgb::new(0.96, 0.80, 0.10, None)),   // gold
        "ForkEvent" => Color::Rgb(Rgb::new(0.27, 0.68, 0.96, None)),    // blue
        "CreateEvent" => Color::Rgb(Rgb::new(0.13, 0.78, 0.47, None)),  // green
        "DeleteEvent" => Color::Rgb(Rgb::new(0.78, 0.25, 0.25, None)),  // red
        _ => Color::Rgb(Rgb::new(0.50, 0.50, 0.50, None)),
    }
}

struct EventDescription {
    main: String,
    detail: Vec<String>,
}

fn describe_event(event: &GitHubEvent) -> EventDescription {
    let repo = &event.repo.name;
    let p = &event.payload;

    let (main, detail) = match event.kind.as_str() {
        "PushEvent" => {
            let branch = p["ref"]
                .as_str()
                .unwrap_or("")
                .trim_start_matches("refs/heads/");
            let commits_arr = p["commits"].as_array();
            let count = p["size"]
                .as_u64()
                .map(|n| n as usize)
                .filter(|&n| n > 0)
                .unwrap_or_else(|| commits_arr.map(|c| c.len()).unwrap_or(0));
            let main = if count > 0 {
                let label = if count == 1 { "commit" } else { "commits" };
                format!("Pushed {count} {label} to {repo} ({branch})")
            } else {
                format!("Pushed to {repo} ({branch})")
            };
            let detail: Vec<String> = commits_arr
                .into_iter()
                .flatten()
                .take(5)
                .filter_map(|c| c["message"].as_str())
                .map(|m| format!("  {}", m.lines().next().unwrap_or(m)))
                .collect();
            (main, detail)
        }
        "PullRequestEvent" => {
            let action = p["action"].as_str().unwrap_or("updated");
            let merged =
                action == "closed" && p["pull_request"]["merged"].as_bool().unwrap_or(false);
            let label = if merged { "merged" } else { action };
            let title = p["pull_request"]["title"].as_str().unwrap_or("");
            let number = p["pull_request"]["number"].as_u64().unwrap_or(0);
            let detail = match (
                p["pull_request"]["additions"].as_u64(),
                p["pull_request"]["deletions"].as_u64(),
                p["pull_request"]["changed_files"].as_u64(),
            ) {
                (Some(a), Some(d), Some(f)) => {
                    let fword = if f == 1 { "file" } else { "files" };
                    vec![format!("    +{a} \u{2212}{d} across {f} {fword}")]
                }
                _ => vec![],
            };
            (
                format!("{} PR #{number}: {title} in {repo}", capitalise(label)),
                detail,
            )
        }
        "IssuesEvent" => {
            let action = p["action"].as_str().unwrap_or("updated");
            let title = p["issue"]["title"].as_str().unwrap_or("");
            let number = p["issue"]["number"].as_u64().unwrap_or(0);
            (
                format!("{} issue #{number}: {title} in {repo}", capitalise(action)),
                vec![],
            )
        }
        "IssueCommentEvent" => {
```

```
            let title = p["issue"]["title"].as_str().unwrap_or("");
            let number = p["issue"]["number"].as_u64().unwrap_or(0);
            (
                format!("Commented on issue #{number}: {title} in {repo}"),
                vec![],
            )
        }
        "PullRequestReviewEvent" => {
            let state = p["review"]["state"].as_str().unwrap_or("reviewed");
            let number = p["pull_request"]["number"].as_u64().unwrap_or(0);
            (
                format!("{} PR #{number} in {repo}", capitalise(state)),
                vec![],
            )
        }
        "PullRequestReviewCommentEvent" => {
            let number = p["pull_request"]["number"].as_u64().unwrap_or(0);
            (format!("Reviewed PR #{number} in {repo}"), vec![])
        }
        "CreateEvent" => {
            let ref_type = p["ref_type"].as_str().unwrap_or("ref");
            let ref_name = p["ref"].as_str().unwrap_or("");
            if ref_name.is_empty() {
                (format!("Created {ref_type} {repo}"), vec![])
            } else {
                (format!("Created {ref_type} '{ref_name}' in {repo}"), vec![])
            }
        }
        "DeleteEvent" => {
            let ref_type = p["ref_type"].as_str().unwrap_or("ref");
            let ref_name = p["ref"].as_str().unwrap_or("");
            (format!("Deleted {ref_type} '{ref_name}' in {repo}"), vec![])
        }
        "ForkEvent" => {
            let forkee = p["forkee"]["full_name"].as_str().unwrap_or(repo);
            (format!("Forked {repo} \u{2192} {forkee}"), vec![])
        }
        "WatchEvent" => (format!("Starred {repo}"), vec![]),
        "ReleaseEvent" => {
            let action = p["action"].as_str().unwrap_or("published");
            let tag = p["release"]["tag_name"].as_str().unwrap_or("");
            (
                format!("{} release {tag} in {repo}", capitalise(action)),
                vec![],
            )
        }
        "CommitCommentEvent" => (format!("Commented on a commit in {repo}"), vec![]),
        "GollumEvent" => (format!("Updated wiki in {repo}"), vec![]),
        "MemberEvent" => {
            let action = p["action"].as_str().unwrap_or("updated");
            let member = p["member"]["login"].as_str().unwrap_or("someone");
            (
                format!("{} {member} as collaborator in {repo}", capitalise(action)),
                vec![],
            )
        }
        "PublicEvent" => (format!("Made {repo} public"), vec![]),
        "SponsorshipEvent" => (format!("Sponsorship activity in {repo}"), vec![]),
        other => (format!("{other} in {repo}"), vec![]),
    };

    EventDescription { main, detail }
}

/// Returns the most relevant clickable URL for a GitHub event, if one exists.
fn event_url(event: &GitHubEvent) -> Option<String> {
    let repo = &event.repo.name;
    let p = &event.payload;
    match event.kind.as_str() {
        "PushEvent" => {
            if let Some(sha) = p["head"].as_str() {
                Some(format!("https://github.com/{repo}/commit/{sha}"))
            } else {
                p["ref"].as_str().map(|r| {
                    let branch = r.trim_start_matches("refs/heads/");
                    format!("https://github.com/{repo}/tree/{branch}")
                })
            }
```

```
        }
        "PullRequestEvent" => p["pull_request"]["html_url"].as_str().map(str::to_string),
        "IssuesEvent" => p["issue"]["html_url"].as_str().map(str::to_string),
        "IssueCommentEvent" => p["comment"]["html_url"].as_str().map(str::to_string),
        "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => {
            p["pull_request"]["html_url"].as_str().map(str::to_string)
        }
        "ForkEvent" => p["forkee"]["html_url"].as_str().map(str::to_string),
        "ReleaseEvent" => p["release"]["html_url"].as_str().map(str::to_string),
        _ => Some(format!("https://github.com/{repo}")),
    }
}

fn capitalise(s: &str) -> String {
    let mut chars = s.chars();
    match chars.next() {
        None => String::new(),
        Some(c) => c.to_uppercase().to_string() + chars.as_str(),
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::pdf;
    use crate::types::Config;

    fn push_event() -> GitHubEvent {
        GitHubEvent {
            kind: "PushEvent".to_string(),
            repo: crate::github::EventRepo {
                name: "alice/myrepo".to_string(),
            },
            payload: serde_json::json!({
                "ref": "refs/heads/main",
                "commits": [
                    { "message": "fix: correct typo" },
                    { "message": "feat: add feature" }
                ]
            }),
            created_at: "2024-03-01T12:00:00Z".to_string(),
        }
    }

    fn pr_event() -> GitHubEvent {
        GitHubEvent {
            kind: "PullRequestEvent".to_string(),
            repo: crate::github::EventRepo {
                name: "alice/myrepo".to_string(),
            },
            payload: serde_json::json!({
                "action": "opened",
                "pull_request": { "number": 42, "title": "Add dark mode" }
            }),
            created_at: "2024-03-01T11:00:00Z".to_string(),
        }
    }

    #[test]
    fn render_activity_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(
            &mut builder,
            &[push_event(), pr_event()],
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_activity_empty_is_noop() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
```

```
        let page_before = builder.current_page();
        super::render(&mut builder, &[], &std::collections::HashMap::new());
        assert_eq!(builder.current_page(), page_before);
    }

    #[test]
    fn capitalise_works() {
        assert_eq!(super::capitalise("opened"), "Opened");
        assert_eq!(super::capitalise(""), "");
        assert_eq!(super::capitalise("x"), "X");
    }

    #[test]
    fn describe_push_event() {
        let desc = super::describe_event(&push_event());
        assert!(desc.main.contains("Pushed 2 commits"));
        assert!(desc.main.contains("alice/myrepo"));
        assert!(desc.main.contains("main"));
        assert_eq!(desc.detail.len(), 2);
    }

    #[test]
    fn describe_pr_event() {
        let desc = super::describe_event(&pr_event());
        assert!(desc.main.contains("Opened PR #42"));
        assert!(desc.main.contains("Add dark mode"));
    }

    #[test]
    fn event_icons_cover_all_known_types() {
        [
            "PushEvent",
            "PullRequestEvent",
            "IssuesEvent",
            "ReleaseEvent",
            "WatchEvent",
            "IssueCommentEvent",
            "PullRequestReviewEvent",
            "PullRequestReviewCommentEvent",
            "ForkEvent",
            "CreateEvent",
            "DeleteEvent",
            "UnknownEvent",
        ]
        .iter()
        .for_each(|kind| {
            let e = GitHubEvent {
                kind: kind.to_string(),
                repo: crate::github::EventRepo {
                    name: "a/b".to_string(),
                },
                payload: serde_json::json!({}),
                created_at: "2024-01-01T00:00:00Z".to_string(),
            };
            assert!(!super::event_icon(&e).is_empty());
            // icon color must not panic
            let _ = super::event_icon_color(&e);
        });
    }

    fn make_event(kind: &str, payload: serde_json::Value) -> GitHubEvent {
        GitHubEvent {
            kind: kind.to_string(),
            repo: crate::github::EventRepo {
                name: "alice/repo".to_string(),
            },
            payload,
            created_at: "2024-03-01T09:30:00Z".to_string(),
        }
    }

    #[test]
    fn describe_issue_comment_event() {
        let e = make_event(
            "IssueCommentEvent",
            serde_json::json!({ "issue": { "number": 7, "title": "Bug" } }),
        );
        let d = super::describe_event(&e);
```

```
        assert!(d.main.contains("#7"));
        assert!(d.main.contains("Bug"));
    }

    #[test]
    fn describe_pr_review_event() {
        let e = make_event(
            "PullRequestReviewEvent",
            serde_json::json!({ "review": { "state": "approved" }, "pull_request": { "number": 3 } }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("Approved"));
        assert!(d.main.contains("#3"));
    }

    #[test]
    fn describe_pr_review_comment_event() {
        let e = make_event(
            "PullRequestReviewCommentEvent",
            serde_json::json!({ "pull_request": { "number": 5 } }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("#5"));
    }

    #[test]
    fn describe_create_event_with_ref() {
        let e = make_event(
            "CreateEvent",
            serde_json::json!({ "ref_type": "branch", "ref": "feature/x" }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("branch"));
        assert!(d.main.contains("feature/x"));
    }

    #[test]
    fn describe_create_event_no_ref() {
        let e = make_event(
            "CreateEvent",
            serde_json::json!({ "ref_type": "repository", "ref": "" }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("repository"));
        assert!(!d.main.contains("''"));
    }

    #[test]
    fn describe_delete_event() {
        let e = make_event(
            "DeleteEvent",
            serde_json::json!({ "ref_type": "branch", "ref": "old-feature" }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("old-feature"));
    }

    #[test]
    fn describe_fork_event() {
        let e = make_event(
            "ForkEvent",
            serde_json::json!({ "forkee": { "full_name": "bob/repo" } }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("bob/repo"));
    }

    #[test]
    fn describe_watch_event() {
        let d = super::describe_event(&make_event("WatchEvent", serde_json::json!({})));
        assert!(d.main.contains("Starred"));
    }

    #[test]
    fn describe_release_event() {
        let e = make_event(
            "ReleaseEvent",
            serde_json::json!({ "action": "published", "release": { "tag_name": "v1.2.3" } }),
```

```rust
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("v1.2.3"));
    }

    #[test]
    fn describe_commit_comment_event() {
        let d = super::describe_event(&make_event("CommitCommentEvent", serde_json::json!({})));
        assert!(d.main.contains("commit"));
    }

    #[test]
    fn describe_gollum_event() {
        let d = super::describe_event(&make_event("GollumEvent", serde_json::json!({})));
        assert!(d.main.contains("wiki"));
    }

    #[test]
    fn describe_member_event() {
        let e = make_event(
            "MemberEvent",
            serde_json::json!({ "action": "added", "member": { "login": "bob" } }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("bob"));
    }

    #[test]
    fn describe_public_event() {
        let d = super::describe_event(&make_event("PublicEvent", serde_json::json!({})));
        assert!(d.main.contains("public"));
    }

    #[test]
    fn describe_unknown_event() {
        let d = super::describe_event(&make_event("CoolNewEvent", serde_json::json!({})));
        assert!(!d.main.is_empty());
    }

    #[test]
    fn describe_pr_event_merged() {
        let e = make_event(
            "PullRequestEvent",
            serde_json::json!({
                "action": "closed",
                "pull_request": {
                    "number": 10, "title": "Big feature",
                    "merged": true,
                    "additions": 50, "deletions": 5, "changed_files": 3
                }
            }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("Merged"));
        assert_eq!(d.detail.len(), 1);
    }

    #[test]
    fn describe_push_event_no_size_field() {
        // When "size" is absent, falls back to commits array length.
        let e = make_event(
            "PushEvent",
            serde_json::json!({
                "ref": "refs/heads/main",
                "commits": [{ "message": "only commit" }]
            }),
        );
        let d = super::describe_event(&e);
        assert!(d.main.contains("1 commit"));
    }

    #[test]
    fn event_url_push_without_head_uses_branch() {
        let e = make_event("PushEvent", serde_json::json!({ "ref": "refs/heads/feat" }));
        let url = super::event_url(&e).unwrap();
        assert!(url.contains("feat"));
    }

```

```rust
    #[test]
    fn event_url_pull_request_uses_html_url() {
        let e = make_event(
            "PullRequestEvent",
            serde_json::json!({ "pull_request": { "html_url": "https://github.com/alice/repo/pull/1" } }),
        );
        assert_eq!(
            super::event_url(&e),
            Some("https://github.com/alice/repo/pull/1".to_string())
        );
    }

    #[test]
    fn event_url_issues_event() {
        let e = make_event(
            "IssuesEvent",
            serde_json::json!({ "issue": { "html_url": "https://github.com/alice/repo/issues/2" } }),
        );
        assert_eq!(
            super::event_url(&e),
            Some("https://github.com/alice/repo/issues/2".to_string())
        );
    }

    #[test]
    fn event_url_fork_event() {
        let e = make_event(
            "ForkEvent",
            serde_json::json!({ "forkee": { "html_url": "https://github.com/bob/repo" } }),
        );
        assert_eq!(
            super::event_url(&e),
            Some("https://github.com/bob/repo".to_string())
        );
    }

    #[test]
    fn event_url_catchall_returns_repo_url() {
        let url = super::event_url(&make_event("WatchEvent", serde_json::json!({}))).unwrap();
        assert!(url.contains("alice/repo"));
    }

    #[test]
    fn render_activity_many_event_types() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let events: Vec<GitHubEvent> = [
            "IssuesEvent",
            "IssueCommentEvent",
            "PullRequestReviewEvent",
            "CreateEvent",
            "DeleteEvent",
            "ForkEvent",
            "ReleaseEvent",
            "WatchEvent",
        ]
        .iter()
        .map(|kind| make_event(kind, serde_json::json!({})))
        .collect();
        super::render(&mut builder, &events, &std::collections::HashMap::new());
        assert!(!builder.finish().is_empty());
    }
}
```

src/pdf/user_cover.rs

```rust
use printpdf::{Actions, Color, Pt, Rgb};

use super::layout::{PageBuilder, Span};
use crate::github::GitHubUser;

const CRATES_URL: &str = "https://crates.io/crates/gitprint";
const LABEL_COL: usize = 14;
const CHAR_WIDTH: f32 = 0.6;

fn separator_line(width_pt: f32, font_size: f32) -> String {
    let chars = (width_pt / (font_size * CHAR_WIDTH)).max(1.0) as usize;
    "─".repeat(chars)
}

/// Word-wrap `text` into lines of at most `max_chars` characters, breaking at word boundaries.
fn word_wrap(text: &str, max_chars: usize) -> Vec<String> {
    if max_chars == 0 {
        return vec![text.to_string()];
    }
    let (mut lines, last) = text.split_whitespace().fold(
        (Vec::<String>::new(), String::new()),
        |(mut lines, mut cur), word| {
            if !cur.is_empty() && cur.len() + 1 + word.len() > max_chars {
                lines.push(std::mem::take(&mut cur));
            } else if !cur.is_empty() {
                cur.push(' ');
            }
            cur.push_str(word);
            (lines, cur)
        },
    );
    if !last.is_empty() {
        lines.push(last);
    }
    lines
}

/// Renders the user report cover page with profile info, metadata table, and footer.
pub fn render(builder: &mut PageBuilder, user: &GitHubUser, total_stars: u64) {
    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
    let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
    let lh = builder.line_height();

    const TABLE_SIZE: f32 = 9.0;

    let display_name = user.name.as_deref().unwrap_or(&user.login);

    // ── Title ──────────────────────────────────────────────────────
    builder.vertical_space(120.0);
    builder.write_centered(display_name, &bold, Pt(28.0), black.clone());
    builder.add_link(28.0 + 4.0, Actions::Uri(user.html_url.clone()));

    if display_name != user.login {
        builder.vertical_space(6.0);
        builder.write_centered(
            &format!("@{}", user.login),
            &regular,
            Pt(12.0),
            gray.clone(),
        );
        builder.add_link(12.0 + 4.0, Actions::Uri(user.html_url.clone()));
    }

    builder.vertical_space(32.0);

    // ── Metadata table ─────────────────────────────────────────────
    builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.75, 0.75, 0.75, None)), 0.5);
    builder.vertical_space(8.0);

    // Build rows dynamically — only show non-empty fields.
    let repos_str = user.public_repos.to_string();
    let stars_str = total_stars.to_string();
    let followers_str = user.followers.to_string();
    let following_str = user.following.to_string();
    let member_since = user
```

```
        .created_at
        .get(..10)
        .unwrap_or(&user.created_at)
        .to_string();

    let value_col_max_chars = ((builder.usable_width_pt()
        - LABEL_COL as f32 * TABLE_SIZE * CHAR_WIDTH)
        / (TABLE_SIZE * CHAR_WIDTH))
        .max(1.0) as usize;

    let email_url = user.email.as_ref().map(|e| format!("mailto:{e}"));
    let repos_url = format!("{}?tab=repositories", user.html_url);
    let followers_url = format!("{}?tab=followers", user.html_url);
    let following_url = format!("{}?tab=following", user.html_url);

    [
        ("Bio", user.bio.as_deref().unwrap_or(""), None::<String>),
        ("Location", user.location.as_deref().unwrap_or(""), None),
        ("Company", user.company.as_deref().unwrap_or(""), None),
        (
            "Blog",
            user.blog.as_deref().unwrap_or(""),
            user.blog.as_ref().map(|b| {
                if b.starts_with("http") {
                    b.clone()
                } else {
                    format!("https://{b}")
                }
            }),
        ),
        (
            "Email",
            user.email.as_deref().unwrap_or(""),
            email_url.clone(),
        ),
        ("Public Repos", &repos_str, Some(repos_url.clone())),
        ("Total Stars", &stars_str, None),
        ("Followers", &followers_str, Some(followers_url.clone())),
        ("Following", &following_str, Some(following_url.clone())),
        ("Member Since", &member_since, None),
        ("Profile", &user.html_url, Some(user.html_url.clone())),
    ]
    .into_iter()
    .filter(|(_, value, _)| !value.is_empty())
    .for_each(|(label, value, url)| {
        word_wrap(value, value_col_max_chars)
            .into_iter()
            .enumerate()
            .for_each(|(i, line)| {
                let label_text = if i == 0 {
                    format!("{label:<LABEL_COL$}")
                } else {
                    " ".repeat(LABEL_COL)
                };
                builder.write_line(&[
                    Span {
                        text: label_text,
                        font_id: bold.clone(),
                        size: Pt(TABLE_SIZE),
                        color: black.clone(),
                    },
                    Span {
                        text: line,
                        font_id: regular.clone(),
                        size: Pt(TABLE_SIZE),
                        color: black.clone(),
                    },
                ]);
            });
        if let Some(u) = url {
            builder.add_link(lh, Actions::Uri(u));
        }
    });

    builder.vertical_space(4.0);
    builder.draw_horizontal_rule(Color::Rgb(Rgb::new(0.75, 0.75, 0.75, None)), 0.5);

    // ── Footer ──────────────────────────────────────────────────────────
```

```rust
    let version = env!("CARGO_PKG_VERSION");
    let footer_text =
        format!("Generated with gitprint v{version} ({CRATES_URL}), a Izel Nakri production");
    let footer_size = Pt(7.0);
    let footer_area = lh + 4.0 + footer_size.0 + 4.0;
    builder.vertical_space((builder.remaining_pt() - footer_area).max(0.0));

    builder.write_line(&[Span {
        text: separator_line(builder.usable_width_pt(), footer_size.0),
        font_id: regular.clone(),
        size: footer_size,
        color: gray.clone(),
    }]);
    builder.vertical_space(4.0);
    builder.write_centered(&footer_text, &regular, footer_size, gray);
    builder.add_link(footer_size.0 + 4.0, Actions::Uri(CRATES_URL.to_string()));

    builder.page_break();
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::pdf;
    use crate::types::{Config, UserReportConfig};

    fn test_user() -> GitHubUser {
        GitHubUser {
            login: "alice".to_string(),
            name: Some("Alice Dev".to_string()),
            bio: Some("Rust enthusiast".to_string()),
            location: Some("Berlin".to_string()),
            company: Some("Acme Corp".to_string()),
            blog: Some("https://alice.dev".to_string()),
            email: Some("alice@example.com".to_string()),
            public_repos: 42,
            followers: 100,
            following: 50,
            created_at: "2018-03-15T10:00:00Z".to_string(),
            html_url: "https://github.com/alice".to_string(),
        }
    }

    fn test_user_config() -> UserReportConfig {
        UserReportConfig {
            username: "alice".to_string(),
            output_path: "/tmp/alice.pdf".into(),
            paper_size: crate::types::PaperSize::A4,
            landscape: false,
            last_repos: 5,
            last_commits: 5,
            no_diffs: false,
            font_size: 8.0,
            github_token: None,
            since: None,
            until: None,
            activity: crate::types::ActivityFilter::All,
            events: 30,
        }
    }

    #[test]
    fn render_user_cover_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(&mut builder, &test_user(), 1337);
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_user_cover_no_name() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut user = test_user();
```

```
        user.name = None;
        super::render(&mut builder, &user, 0);
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_user_cover_minimal_fields() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let uc = test_user_config();
        let mut builder = pdf::create_user_builder(&uc, fonts);
        let user = GitHubUser {
            login: "bob".to_string(),
            name: None,
            bio: None,
            location: None,
            company: None,
            blog: None,
            email: None,
            public_repos: 1,
            followers: 0,
            following: 0,
            created_at: "2020-01-01T00:00:00Z".to_string(),
            html_url: "https://github.com/bob".to_string(),
        };
        super::render(&mut builder, &user, 0);
        assert!(!builder.finish().is_empty());
    }
}
```

src/pdf/user_repos.rs

```rust
use std::collections::HashMap;

use printpdf::{Actions, Color, Pt, Rgb};

use super::layout::{PageBuilder, Span};
use crate::github::{GitHubEvent, GitHubRepo};

/// Renders a titled section listing repositories with stats and recent activity context.
pub fn render(
    builder: &mut PageBuilder,
    title: &str,
    repos: &[GitHubRepo],
    events: &[GitHubEvent],
    commit_msgs: &std::collections::HashMap<String, String>,
) {
    if repos.is_empty() {
        return;
    }

    // Index events by repo name, keeping the newest (API returns newest-first).
    let push_ctx: HashMap<&str, &GitHubEvent> = events
        .iter()
        .filter(|e| e.kind == "PushEvent")
        .fold(HashMap::new(), |mut map, e| {
            map.entry(e.repo.name.as_str()).or_insert(e);
            map
        });
    let activity_ctx: HashMap<&str, &GitHubEvent> = events
        .iter()
        .filter(|e| e.kind != "PushEvent")
        .fold(HashMap::new(), |mut map, e| {
            map.entry(e.repo.name.as_str()).or_insert(e);
            map
        });

    let bold = builder.font(true, false).clone();
    let regular = builder.font(false, false).clone();
    let italic = builder.font(false, true).clone();
    let black = Color::Rgb(Rgb::new(0.0, 0.0, 0.0, None));
    let gray = Color::Rgb(Rgb::new(0.47, 0.47, 0.47, None));
    let dark_gray = Color::Rgb(Rgb::new(0.25, 0.25, 0.25, None));
    let gold = Color::Rgb(Rgb::new(0.90, 0.72, 0.10, None));
    let rule_gray = Color::Rgb(Rgb::new(0.85, 0.85, 0.85, None));

    builder.ensure_space(builder.line_height() * 3.0);
    builder.write_centered(title, &bold, Pt(14.0), black.clone());
    builder.vertical_space(8.0);
    builder.draw_horizontal_rule(rule_gray.clone(), 0.5);
    builder.vertical_space(8.0);

    repos.iter().enumerate().for_each(|(idx, repo)| {
        // Thin separator between repo entries (not before the first one).
        if idx > 0 {
            builder.vertical_space(2.0);
            builder.draw_horizontal_rule(rule_gray.clone(), 0.3);
            builder.vertical_space(8.0);
        }

        builder.ensure_space(builder.line_height() * 5.0);

        // ── Row 1: name (left) + stats (right) ──────────────────────────
        let fork_tag = if repo.fork { " [fork]" } else { "" };
        let lang = repo.language.as_deref().unwrap_or("—");
        let stats = format!(
            "\u{25C6} {} \u{00B7} \u{25B7} {} \u{00B7} ! {} \u{00B7} {}",
            repo.stargazers_count, repo.forks_count, repo.open_issues_count, lang
        );
        builder.write_line_justified(
            &[Span {
                text: format!("{}{fork_tag}", repo.name),
                font_id: bold.clone(),
                size: Pt(9.0),
                color: black.clone(),
            }],
            &[Span {
                text: stats,
                font_id: regular.clone(),
```

```
            size: Pt(8.0),
            color: gold.clone(),
        }],
    );
    builder.add_link(builder.line_height(), Actions::Uri(repo.html_url.clone()));

    // ── Row 2: description ──────────────────────────────────────
    if let Some(desc) = repo.description.as_deref().filter(|d| !d.is_empty()) {
        builder.write_line(&[Span {
            text: format!("  {desc}"),
            font_id: italic.clone(),
            size: Pt(8.0),
            color: gray.clone(),
        }]);
    }

    // ── Row 3: dates + size ─────────────────────────────────────
    let pushed = repo
        .pushed_at
        .as_deref()
        .or(repo.updated_at.as_deref())
        .and_then(|d| d.get(..10))
        .unwrap_or("—");
    let created = repo
        .created_at
        .as_deref()
        .and_then(|d| d.get(..10))
        .unwrap_or("—");
    let size_part = match repo.size {
        0 => String::new(),
        kb if kb < 1024 => format!("  ·  {kb} KB"),
        kb => format!("  ·  {:.1} MB", kb as f64 / 1024.0),
    };
    builder.write_line(&[Span {
        text: format!("  last push {pushed}  ·  created {created}{size_part}"),
        font_id: regular.clone(),
        size: Pt(7.5),
        color: gray.clone(),
    }]);

    // ── Row 4: your recent activity context ─────────────────────
    // Push event → show branch + commit messages (you pushed code here).
    // Non-push event → show what you did (opened issue, reviewed PR, etc.).
    if let Some(ev) = push_ctx.get(repo.full_name.as_str()) {
        let branch = ev.payload["ref"]
            .as_str()
            .unwrap_or("")
            .trim_start_matches("refs/heads/");
        // Prefer a direct link to the HEAD commit; fall back to the branch tree.
        let push_url = ev.payload["head"]
            .as_str()
            .map(|sha| format!("https://github.com/{}/commit/{sha}", repo.full_name))
            .unwrap_or_else(|| format!("https://github.com/{}/tree/{branch}", repo.full_name));
        // Commit messages from the event payload (present for normal pushes).
        let from_payload: Vec<String> = ev.payload["commits"]
            .as_array()
            .into_iter()
            .flatten()
            .filter_map(|c| c["message"].as_str())
            .map(|m| m.lines().next().unwrap_or(m).to_string())
            .take(2)
            .collect();
        // Fall back to API-fetched message via HEAD SHA (force push / rebase gave empty payload).
        let commits = if from_payload.is_empty() {
            ev.payload["head"]
                .as_str()
                .and_then(|sha| commit_msgs.get(sha))
                .map(|msg| vec![msg.clone()])
                .unwrap_or_default()
        } else {
            from_payload
        };
        if commits.is_empty() {
            let date = ev.created_at.get(..10).unwrap_or(&ev.created_at);
            builder.write_line(&[Span {
                text: format!("  \u{2192} pushed to {branch} on {date}"),
                font_id: italic.clone(),
                size: Pt(7.5),
```

```
                    color: dark_gray.clone(),
                }]);
                builder.add_link(builder.line_height(), Actions::Uri(push_url));
            } else {
                builder.write_line(&[Span {
                    text: format!("  \u{2192} pushed to {branch}:"),
                    font_id: italic.clone(),
                    size: Pt(7.5),
                    color: dark_gray.clone(),
                }]);
                builder.add_link(builder.line_height(), Actions::Uri(push_url.clone()));
                commits.iter().for_each(|msg| {
                    builder.write_line(&[Span {
                        text: format!("      {msg}"),
                        font_id: italic.clone(),
                        size: Pt(7.5),
                        color: gray.clone(),
                    }]);
                    builder.add_link(builder.line_height(), Actions::Uri(push_url.clone()));
                });
            }
        } else if let Some(ev) = activity_ctx.get(repo.full_name.as_str()) {
            let date = ev.created_at.get(..10).unwrap_or(&ev.created_at);
            builder.write_line(&[Span {
                text: format!("  \u{2192} {} on {date}", brief_activity(ev)),
                font_id: italic.clone(),
                size: Pt(7.5),
                color: dark_gray.clone(),
            }]);
            builder.add_link(builder.line_height(), Actions::Uri(repo.html_url.clone()));
        }

        builder.vertical_space(4.0);
    });

    builder.vertical_space(12.0);
}

/// One-line description of a non-push GitHub event for display in the repo context row.
fn brief_activity(event: &GitHubEvent) -> String {
    let p = &event.payload;
    match event.kind.as_str() {
        "PullRequestEvent" => {
            let action = p["action"].as_str().unwrap_or("updated");
            let n = p["pull_request"]["number"].as_u64().unwrap_or(0);
            let merged =
                action == "closed" && p["pull_request"]["merged"].as_bool().unwrap_or(false);
            if merged {
                format!("merged PR #{n}")
            } else {
                format!("{action} PR #{n}")
            }
        }
        "IssuesEvent" => {
            let action = p["action"].as_str().unwrap_or("updated");
            let n = p["issue"]["number"].as_u64().unwrap_or(0);
            format!("{action} issue #{n}")
        }
        "IssueCommentEvent" => {
            let n = p["issue"]["number"].as_u64().unwrap_or(0);
            format!("commented on issue #{n}")
        }
        "PullRequestReviewEvent" | "PullRequestReviewCommentEvent" => {
            let n = p["pull_request"]["number"].as_u64().unwrap_or(0);
            format!("reviewed PR #{n}")
        }
        "WatchEvent" => "starred".to_string(),
        "ForkEvent" => "forked".to_string(),
        "ReleaseEvent" => {
            let tag = p["release"]["tag_name"].as_str().unwrap_or("");
            format!("released {tag}")
        }
        "CreateEvent" => {
            let ref_type = p["ref_type"].as_str().unwrap_or("ref");
            let ref_name = p["ref"].as_str().unwrap_or("");
            if ref_name.is_empty() {
                format!("created {ref_type}")
            } else {
```

```
                format!("created {ref_type} '{ref_name}'")
            }
        }
        other => other.replace("Event", "").to_lowercase(),
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::pdf;
    use crate::types::Config;

    fn test_repo(name: &str, stars: u64) -> GitHubRepo {
        GitHubRepo {
            name: name.to_string(),
            full_name: format!("alice/{name}"),
            html_url: format!("https://github.com/alice/{name}"),
            description: Some(format!("{name} — a great project")),
            language: Some("Rust".to_string()),
            stargazers_count: stars,
            forks_count: 10,
            open_issues_count: 3,
            size: 2048,
            pushed_at: Some("2024-03-01T00:00:00Z".to_string()),
            updated_at: Some("2024-03-02T00:00:00Z".to_string()),
            created_at: Some("2020-06-15T00:00:00Z".to_string()),
            fork: false,
        }
    }

    fn test_push_event(repo: &str, branch: &str, msgs: &[&str]) -> GitHubEvent {
        use crate::github::EventRepo;
        let commits: Vec<serde_json::Value> = msgs
            .iter()
            .map(|m| serde_json::json!({ "message": m }))
            .collect();
        GitHubEvent {
            kind: "PushEvent".to_string(),
            repo: EventRepo {
                name: repo.to_string(),
            },
            payload: serde_json::json!({
                "ref": format!("refs/heads/{branch}"),
                "commits": commits,
                "size": msgs.len()
            }),
            created_at: "2024-03-01T09:00:00Z".to_string(),
        }
    }

    #[test]
    fn render_repos_does_not_panic() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        super::render(
            &mut builder,
            "Top Starred Repositories",
            &[test_repo("gitprint", 500), test_repo("another", 200)],
            &[],
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_repos_empty_is_noop() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let page_before = builder.current_page();
        super::render(
            &mut builder,
            "Top Repos",
            &[],
```

```
            &[],
            &std::collections::HashMap::new(),
        );
        assert_eq!(builder.current_page(), page_before);
    }

    #[test]
    fn render_fork_repo_shows_tag() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut repo = test_repo("forked", 5);
        repo.fork = true;
        super::render(
            &mut builder,
            "Forks",
            &[repo],
            &[],
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }

    fn test_issue_event(repo: &str, number: u64) -> GitHubEvent {
        use crate::github::EventRepo;
        GitHubEvent {
            kind: "IssuesEvent".to_string(),
            repo: EventRepo {
                name: repo.to_string(),
            },
            payload: serde_json::json!({ "action": "opened", "issue": { "number": number } }),
            created_at: "2024-03-02T10:00:00Z".to_string(),
        }
    }

    #[test]
    fn render_repos_with_activity_event_context() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let events = [test_issue_event("alice/gitprint", 42)];
        super::render(
            &mut builder,
            "Repos You Were Active In",
            &[test_repo("gitprint", 100)],
            &events,
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_repos_with_push_event_context() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let events = [test_push_event(
            "alice/gitprint",
            "main",
            &["fix: typo", "feat: add feature"],
        )];
        super::render(
            &mut builder,
            "Recently Pushed",
            &[test_repo("gitprint", 100)],
            &events,
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }

    #[test]
    fn render_repos_push_event_no_commits_shows_branch() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
```

```rust
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        // Push with empty commits array — falls through to "pushed to {branch} on {date}" path.
        let events = [test_push_event("alice/gitprint", "main", &[])];
        super::render(
            &mut builder,
            "Pushed",
            &[test_repo("gitprint", 100)],
            &events,
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }

    fn make_event(repo: &str, kind: &str, payload: serde_json::Value) -> GitHubEvent {
        use crate::github::EventRepo;
        GitHubEvent {
            kind: kind.to_string(),
            repo: EventRepo {
                name: repo.to_string(),
            },
            payload,
            created_at: "2024-03-01T10:00:00Z".to_string(),
        }
    }

    #[test]
    fn brief_activity_pr_open() {
        let e = make_event(
            "a/b",
            "PullRequestEvent",
            serde_json::json!({ "action": "opened", "pull_request": { "number": 5, "merged": false } }),
        );
        assert_eq!(super::brief_activity(&e), "opened PR #5");
    }

    #[test]
    fn brief_activity_pr_merged() {
        let e = make_event(
            "a/b",
            "PullRequestEvent",
            serde_json::json!({ "action": "closed", "pull_request": { "number": 9, "merged": true } }),
        );
        assert_eq!(super::brief_activity(&e), "merged PR #9");
    }

    #[test]
    fn brief_activity_issues_event() {
        let e = make_event(
            "a/b",
            "IssuesEvent",
            serde_json::json!({ "action": "opened", "issue": { "number": 3 } }),
        );
        assert_eq!(super::brief_activity(&e), "opened issue #3");
    }

    #[test]
    fn brief_activity_issue_comment() {
        let e = make_event(
            "a/b",
            "IssueCommentEvent",
            serde_json::json!({ "issue": { "number": 7 } }),
        );
        assert_eq!(super::brief_activity(&e), "commented on issue #7");
    }

    #[test]
    fn brief_activity_pr_review() {
        let e = make_event(
            "a/b",
            "PullRequestReviewEvent",
            serde_json::json!({ "pull_request": { "number": 2 } }),
        );
        assert_eq!(super::brief_activity(&e), "reviewed PR #2");
    }

    #[test]
    fn brief_activity_watch() {
```

```
        let e = make_event("a/b", "WatchEvent", serde_json::json!({}));
        assert_eq!(super::brief_activity(&e), "starred");
    }

    #[test]
    fn brief_activity_fork() {
        let e = make_event("a/b", "ForkEvent", serde_json::json!({}));
        assert_eq!(super::brief_activity(&e), "forked");
    }

    #[test]
    fn brief_activity_release() {
        let e = make_event(
            "a/b",
            "ReleaseEvent",
            serde_json::json!({ "release": { "tag_name": "v2.0.0" } }),
        );
        assert_eq!(super::brief_activity(&e), "released v2.0.0");
    }

    #[test]
    fn brief_activity_create_with_ref() {
        let e = make_event(
            "a/b",
            "CreateEvent",
            serde_json::json!({ "ref_type": "branch", "ref": "hotfix" }),
        );
        assert_eq!(super::brief_activity(&e), "created branch 'hotfix'");
    }

    #[test]
    fn brief_activity_create_without_ref() {
        let e = make_event(
            "a/b",
            "CreateEvent",
            serde_json::json!({ "ref_type": "repository", "ref": "" }),
        );
        assert_eq!(super::brief_activity(&e), "created repository");
    }

    #[test]
    fn brief_activity_unknown_event() {
        let e = make_event("a/b", "SpookyEvent", serde_json::json!({}));
        assert!(!super::brief_activity(&e).is_empty());
    }

    #[test]
    fn render_repos_no_description() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = pdf::fonts::load_fonts(&mut doc).unwrap();
        let config = Config::test_default();
        let mut builder = pdf::create_builder(&config, fonts);
        let mut repo = test_repo("nodesc", 10);
        repo.description = None;
        super::render(
            &mut builder,
            "Repos",
            &[repo],
            &[],
            &std::collections::HashMap::new(),
        );
        assert!(!builder.finish().is_empty());
    }
}
```

src/preview.rs 1032 LOC · 34.4 KB · 2026-03-04

```rust
//! Terminal preview mode — renders repository or user data to stdout
//! instead of generating a PDF.

use std::collections::BTreeMap;
use std::path::PathBuf;
use std::sync::Arc;

use crate::filter::FileFilter;
use crate::git;
use crate::github::{CommitDetail, GitHubEvent, GitHubRepo};
use crate::types::{Config, UserReportConfig};
use crate::user_report::fetch_data;
use crate::{format_size, format_utc_now};

// ── ANSI helpers ────────────────────────────────────────────────────

struct Ansi {
    color: bool,
}

impl Ansi {
    fn new() -> Self {
        use std::io::IsTerminal;
        Self {
            color: std::io::stdout().is_terminal(),
        }
    }

    fn bold(&self, s: &str) -> String {
        if self.color {
            format!("\x1b[1m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }

    fn dim(&self, s: &str) -> String {
        if self.color {
            format!("\x1b[2m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }

    fn cyan(&self, s: &str) -> String {
        if self.color {
            format!("\x1b[1;36m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }

    fn yellow(&self, s: &str) -> String {
        if self.color {
            format!("\x1b[33m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }

    fn green(&self, s: &str) -> String {
        if self.color {
            format!("\x1b[32m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }

    fn red(&self, s: &str) -> String {
        if self.color {
            format!("\x1b[31m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }

    fn magenta(&self, s: &str) -> String {
```

```
        if self.color {
            format!("\x1b[35m{s}\x1b[0m")
        } else {
            s.to_string()
        }
    }
}

fn divider(a: &Ansi) -> String {
    a.dim(&"─".repeat(64))
}

fn section_header(a: &Ansi, title: &str) {
    println!();
    println!("{}", divider(a));
    println!();
    println!("  {}", a.cyan(title));
    println!();
}

fn box_header(a: &Ansi, title: &str) {
    let inner = format!("  {title}  ");
    let width = 64usize.max(inner.len() + 4);
    let bar = "─".repeat(width - 2);
    println!("{}", a.dim(&format!("┌{bar}┐")));
    println!("{}", a.bold(&format!("│{inner:<pad$}│", pad = width - 2)));
    println!("{}", a.dim(&format!("└{bar}┘")));
}

fn kv(a: &Ansi, key: &str, value: &str) {
    if value.is_empty() {
        return;
    }
    println!("  {}  {}", a.cyan(&format!("{key:<12}")), value);
}

fn format_number(n: usize) -> String {
    let s = n.to_string();
    let mut result = String::new();
    s.chars().rev().enumerate().for_each(|(i, c)| {
        if i > 0 && i % 3 == 0 {
            result.push(',');
        }
        result.push(c);
    });
    result.chars().rev().collect()
}

fn fmt_u64(n: u64) -> String {
    format_number(n as usize)
}

// ── Repository preview ──────────────────────────────────────────────

/// Previews a repository or file in the terminal.
pub async fn repo(config: &Config) -> anyhow::Result<()> {
    let a = Ansi::new();
    let info = git::verify_repo(&config.repo_path).await?;

    // ── Single-file mode ────────────────────────────────────────────
    if let Some(ref single_file) = info.single_file {
        let (content_res, last_modified) = tokio::join!(
            git::read_file_content(&info.root, single_file, config),
            git::file_last_modified(&info.root, single_file, config, info.is_git),
        );
        let content = content_res?;
        let line_count = content.lines().count();
        let size_str = format_size(content.len() as u64);

        box_header(&a, &single_file.display().to_string());
        println!();
        kv(&a, "LINES", &format_number(line_count));
        kv(&a, "SIZE", &size_str);
        kv(&a, "MODIFIED", &last_modified);
        println!();
        println!("{}", divider(&a));
        println!();
        content.lines().enumerate().take(200).for_each(|(i, line)| {
```

```
            println!("  {}  {line}", a.dim(&format!("{:4}", i + 1)));
        });
        if line_count > 200 {
            println!(
                "  {}",
                a.dim(&format!(
                    "  … {} more lines",
                    format_number(line_count - 200)
                ))
            );
        }
        println!();
        return Ok(());
    }

    // ── Multi-file / repository mode ─────────────────────────────────────
    let repo_path = info.root.clone();
    let is_git = info.is_git;
    let scope = info.scope.clone();
    let is_remote = config.remote_url.is_some();
    let generated_at = format_utc_now();
    let repo_path2 = repo_path.clone();
    let config2 = config.clone();
    let repo_path3 = repo_path.clone();

    let (metadata_res, all_paths_res, date_map_res, fs_owner_group, git_repo_size, fs_size) = tokio::join!(
        git::get_metadata(&repo_path, config, is_git, scope.as_deref()),
        git::list_tracked_files(&repo_path, config, is_git, scope.as_deref()),
        git::file_last_modified_dates(&repo_path, config, is_git, scope.as_deref()),
        async {
            if is_remote {
                (None, None)
            } else {
                git::fs_owner_group(&config.repo_path).await
            }
        },
        async {
            if is_git {
                git::git_tracked_size(&repo_path2, &config2).await
            } else {
                String::new()
            }
        },
        async {
            if is_remote {
                String::new()
            } else {
                git::fs_dir_size(&repo_path3).await
            }
        },
    );

    let mut metadata = metadata_res?;
    if let Some(ref url) = config.remote_url {
        metadata.name = git::repo_name_from_url(url);
    }
    metadata.fs_owner = fs_owner_group.0;
    metadata.fs_group = fs_owner_group.1;
    metadata.generated_at = generated_at;
    metadata.repo_size = git_repo_size;
    metadata.fs_size = fs_size;
    if !is_remote {
        metadata.repo_absolute_path = Some(repo_path.clone());
    }

    let date_map = Arc::new(date_map_res?);
    let file_filter = FileFilter::new(&config.include_patterns, &config.exclude_patterns)?;
    let mut paths: Vec<PathBuf> = file_filter.filter_paths(all_paths_res?).collect();
    paths.sort_unstable();

    // Read file contents in parallel to get LOC + size info.
    let mut read_set: tokio::task::JoinSet<Option<(PathBuf, usize, String, String)>> =
        tokio::task::JoinSet::new();
    paths.iter().for_each(|path| {
        let p = path.clone();
        let r = repo_path.clone();
        let c = config.clone();
        let dates = Arc::clone(&date_map);
```

```
        read_set.spawn(async move {
            let content = git::read_file_content(&r, &p, &c).await.ok()?;
            if crate::filter::is_binary(content.as_bytes()) || crate::filter::is_minified(&content)
            {
                return None;
            }
            let line_count = content.lines().count();
            let size_str = format_size(content.len() as u64);
            let last_modified = dates.get(&p).cloned().unwrap_or_default();
            Some((p, line_count, size_str, last_modified))
        });
    });

    let mut files: Vec<(PathBuf, usize, String, String)> =
        read_set.join_all().await.into_iter().flatten().collect();
    files.sort_unstable_by(|a, b| a.0.cmp(&b.0));

    metadata.file_count = files.len();
    metadata.total_lines = files.iter().map(|(_, lc, _, _)| lc).sum();

    let effective_remote = config
        .remote_url
        .as_deref()
        .or(metadata.detected_remote_url.as_deref());

    // ── Header ──────────────────────────────────────────────────────────
    box_header(&a, &metadata.name);
    println!();

    let commit_first_line = metadata.commit_message.lines().next().unwrap_or("");
    let commit_line = format!(
        "{}  ·  {}  ·  {}",
        a.yellow(&metadata.commit_hash_short),
        metadata
            .commit_date
            .get(..10)
            .unwrap_or(&metadata.commit_date),
        commit_first_line,
    );
    kv(&a, "BRANCH", &metadata.branch);
    kv(&a, "COMMIT", &commit_line);
    let author = if metadata.commit_author_email.is_empty() {
        metadata.commit_author.clone()
    } else {
        format!(
            "{}  <{}>",
            metadata.commit_author, metadata.commit_author_email
        )
    };
    kv(&a, "AUTHOR", &author);
    if let Some(url) = effective_remote {
        kv(&a, "REMOTE", url);
    }
    if let Some(path) = &metadata.repo_absolute_path {
        kv(&a, "PATH", &path.display().to_string());
    }
    let size_info = match (metadata.repo_size.is_empty(), metadata.fs_size.is_empty()) {
        (false, false) => {
            format!(
                "{}  (git tracked)  ·  {}  (disk)",
                metadata.repo_size, metadata.fs_size
            )
        }
        (false, true) => metadata.repo_size.clone(),
        (true, false) => metadata.fs_size.clone(),
        _ => String::new(),
    };
    kv(&a, "SIZE", &size_info);
    if let (Some(owner), Some(group)) = (&metadata.fs_owner, &metadata.fs_group) {
        kv(&a, "OWNER", &format!("{owner}:{group}"));
    }
    kv(&a, "GENERATED", &metadata.generated_at);
    println!();
    println!(
        "  {}  {}    {}  {}",
        a.cyan("FILES"),
        a.bold(&format_number(metadata.file_count)),
        a.cyan("LINES"),
```

```rust
        a.bold(&format_number(metadata.total_lines)),
    );

    // ── Directory tree ──────────────────────────────────────────────────
    if config.file_tree {
        section_header(&a, "DIRECTORY TREE");
        let tree_paths: Vec<PathBuf> = files.iter().map(|(p, _, _, _)| p.clone()).collect();
        print_tree(&a, &tree_paths, &metadata.name);
    }

    // ── File list ───────────────────────────────────────────────────────
    section_header(&a, &format!("FILES  ({})", format_number(files.len())));

    let max_path = files
        .iter()
        .map(|(p, _, _, _)| p.display().to_string().len())
        .max()
        .unwrap_or(4)
        .min(60);
    let max_loc = files
        .iter()
        .map(|(_, lc, _, _)| format_number(*lc).len())
        .max()
        .unwrap_or(3)
        .max(3);
    let max_size = files
        .iter()
        .map(|(_, _, s, _)| s.len())
        .max()
        .unwrap_or(4)
        .max(4);

    println!(
        "  {}  {:<path_w$}  {:>loc_w$}  {:<size_w$}  {}",
        a.dim("    "),
        a.dim("PATH"),
        a.dim("LOC"),
        a.dim("SIZE"),
        a.dim("MODIFIED"),
        path_w = max_path,
        loc_w = max_loc,
        size_w = max_size,
    );
    println!(
        "  {}",
        a.dim(&"─".repeat(max_path + max_loc + max_size + 26))
    );

    files
        .iter()
        .enumerate()
        .for_each(|(i, (path, line_count, size_str, last_modified))| {
            println!(
                "  {}  {:<path_w$}  {:>loc_w$}  {:<size_w$}  {}",
                a.dim(&format!("{:4}.", i + 1)),
                path.display(),
                a.bold(&format_number(*line_count)),
                size_str,
                a.dim(last_modified),
                path_w = max_path,
                loc_w = max_loc,
                size_w = max_size,
            );
        });

    println!();
    Ok(())
}

// ── User report preview ─────────────────────────────────────────────────

/// Previews a GitHub user report in the terminal.
pub async fn user(config: &UserReportConfig) -> anyhow::Result<()> {
    let a = Ansi::new();
    eprintln!("Fetching GitHub data for @{}...", config.username);
    let data = fetch_data(config).await?;

    // ── Header ──────────────────────────────────────────────────────────
```

```rust
    let title = match data.user.name.as_deref().filter(|n| !n.is_empty()) {
        Some(name) => format!("{name}  (@{})", data.user.login),
        None => format!("@{}", data.user.login),
    };
    box_header(&a, &title);
    println!();

    if let Some(ref bio) = data.user.bio {
        kv(&a, "BIO", bio);
    }
    if let Some(ref loc) = data.user.location {
        kv(&a, "LOCATION", loc);
    }
    if let Some(ref company) = data.user.company {
        kv(&a, "COMPANY", company);
    }
    if let Some(blog) = data.user.blog.as_deref().filter(|s| !s.is_empty()) {
        kv(&a, "BLOG", blog);
    }
    if let Some(email) = data.user.email.as_deref().filter(|s| !s.is_empty()) {
        kv(&a, "EMAIL", email);
    }
    kv(&a, "PROFILE", &data.user.html_url);

    let joined = data
        .user
        .created_at
        .get(..10)
        .unwrap_or(&data.user.created_at);
    println!();
    println!(
        "  {}  {}    {}  {}    {}  {}    {}  {}    {}  {}",
        a.cyan("REPOS"),
        a.bold(&fmt_u64(data.user.public_repos)),
        a.cyan("FOLLOWERS"),
        a.bold(&fmt_u64(data.user.followers)),
        a.cyan("FOLLOWING"),
        a.bold(&fmt_u64(data.user.following)),
        a.cyan("STARS"),
        a.bold(&fmt_u64(data.total_stars)),
        a.cyan("JOINED"),
        a.bold(joined),
    );

    // ── Activity feed ──────────────────────────────────────────────
    let display_events = &data.events[..config.events.min(data.events.len())];
    if !display_events.is_empty() {
        section_header(
            &a,
            &format!("RECENT ACTIVITY  ({})", format_number(display_events.len())),
        );
        display_events.iter().for_each(|event| {
            print_event(&a, event, &data.commit_msgs);
        });
    }

    // ── Repositories ──────────────────────────────────────────────
    if !data.starred_repos.is_empty() {
        section_header(&a, "TOP STARRED REPOSITORIES");
        data.starred_repos
            .iter()
            .take(5)
            .for_each(|r| print_repo(&a, r));
    }
    if !data.active_repos.is_empty() {
        section_header(&a, "REPOS YOU WERE ACTIVE IN");
        data.active_repos
            .iter()
            .take(5)
            .for_each(|r| print_repo(&a, r));
    }
    if !data.pushed_repos.is_empty() {
        section_header(&a, "RECENTLY PUSHED TO");
        data.pushed_repos
            .iter()
            .take(config.last_repos)
            .for_each(|r| print_repo(&a, r));
    }
```

```rust

    // ── Commits ──────────────────────────────────────────────────────
    if !data.commit_details.is_empty() {
        let sha_to_branch: std::collections::HashMap<&str, &str> = data
            .events
            .iter()
            .filter(|e| e.kind == "PushEvent")
            .filter_map(|e| {
                let sha = e.payload["head"].as_str()?;
                let branch = e.payload["ref"].as_str()?.trim_start_matches("refs/heads/");
                Some((sha, branch))
            })
            .collect();

        section_header(&a, "RECENT COMMITS");
        data.commit_details.iter().for_each(|(repo, detail)| {
            let branch = sha_to_branch
                .get(detail.sha.as_str())
                .copied()
                .unwrap_or("main");
            print_commit(&a, repo, detail, branch);
        });
    }

    println!();
    Ok(())
}

// ── Event printer ─────────────────────────────────────────────────────

fn print_event(
    a: &Ansi,
    event: &GitHubEvent,
    commit_msgs: &std::collections::HashMap<String, String>,
) {
    let date = event.created_at.get(..10).unwrap_or(&event.created_at);
    let repo = &event.repo.name;

    match event.kind.as_str() {
        "PushEvent" => {
            let branch = event.payload["ref"]
                .as_str()
                .unwrap_or("")
                .trim_start_matches("refs/heads/");
            let head_sha = event.payload["head"].as_str().unwrap_or("");
            let msg = head_sha
                .get(..7)
                .and_then(|short| commit_msgs.get(short).or_else(|| commit_msgs.get(head_sha)))
                .map(|s| s.as_str())
                .unwrap_or("");
            let commit_count = event.payload["commits"]
                .as_array()
                .map(|c| c.len())
                .unwrap_or(0);
            print!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.bold("push   "),
                a.cyan(repo),
                a.dim(&format!("· {branch}")),
            );
            if commit_count > 0 {
                print!("  {}", a.dim(&format!("[{commit_count} commit(s)]")));
            }
            println!();
            if !msg.is_empty() {
                println!(
                    "             {}  {}",
                    a.dim(&head_sha[..7.min(head_sha.len())]),
                    msg
                );
            }
        }
        "PullRequestEvent" => {
            let action = event.payload["action"].as_str().unwrap_or("?");
            let title = event.payload["pull_request"]["title"]
                .as_str()
                .unwrap_or("");
```

```
            let number = event.payload["number"].as_u64().unwrap_or(0);
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.magenta("pr     "),
                a.cyan(repo),
                a.dim(&format!("#{number} {action}")),
            );
            if !title.is_empty() {
                println!("                   {title}");
            }
        }
        "IssuesEvent" => {
            let action = event.payload["action"].as_str().unwrap_or("?");
            let title = event.payload["issue"]["title"].as_str().unwrap_or("");
            let number = event.payload["issue"]["number"].as_u64().unwrap_or(0);
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.yellow("issue  "),
                a.cyan(repo),
                a.dim(&format!("#{number} {action}")),
            );
            if !title.is_empty() {
                println!("                   {title}");
            }
        }
        "IssueCommentEvent" => {
            let number = event.payload["issue"]["number"].as_u64().unwrap_or(0);
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.dim("comment"),
                a.cyan(repo),
                a.dim(&format!("issue #{number}")),
            );
        }
        "CreateEvent" => {
            let ref_type = event.payload["ref_type"].as_str().unwrap_or("?");
            let ref_name = event.payload["ref"].as_str().unwrap_or("");
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.green("create "),
                a.cyan(repo),
                a.dim(&format!("{ref_type} {ref_name}")),
            );
        }
        "DeleteEvent" => {
            let ref_type = event.payload["ref_type"].as_str().unwrap_or("?");
            let ref_name = event.payload["ref"].as_str().unwrap_or("");
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.red("delete "),
                a.cyan(repo),
                a.dim(&format!("{ref_type} {ref_name}")),
            );
        }
        "ForkEvent" => {
            let forkee = event.payload["forkee"]["full_name"].as_str().unwrap_or("");
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.dim("fork   "),
                a.cyan(repo),
                a.dim(&format!("→ {forkee}")),
            );
        }
        "WatchEvent" => {
            println!("  {}  {}  {}", a.dim(date), a.dim("star   "), a.cyan(repo),);
        }
        "ReleaseEvent" => {
            let tag = event.payload["release"]["tag_name"].as_str().unwrap_or("");
            let name = event.payload["release"]["name"].as_str().unwrap_or("");
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
```

```
                a.green("release"),
                a.cyan(repo),
                a.dim(if name.is_empty() { tag } else { name }),
            );
        }
        "PullRequestReviewEvent" => {
            let state = event.payload["review"]["state"].as_str().unwrap_or("?");
            let number = event.payload["pull_request"]["number"]
                .as_u64()
                .unwrap_or(0);
            println!(
                "  {}  {}  {}  {}",
                a.dim(date),
                a.dim("review "),
                a.cyan(repo),
                a.dim(&format!("pr #{number} {state}")),
            );
        }
        other => {
            let label = other.trim_end_matches("Event");
            println!(
                "  {}  {}  {}",
                a.dim(date),
                a.dim(&format!("{label:<7}")),
                a.cyan(repo),
            );
        }
    }
}

// ── Repo printer ──────────────────────────────────────────────────────

fn print_repo(a: &Ansi, repo: &GitHubRepo) {
    let lang = repo.language.as_deref().unwrap_or("—");
    let desc = repo.description.as_deref().unwrap_or("");
    let pushed = repo
        .pushed_at
        .as_deref()
        .and_then(|d| d.get(..10))
        .unwrap_or("");

    println!(
        "  {:<42}  {}  {:>7}  {}  {}",
        a.bold(&repo.full_name),
        a.dim(&format!("{lang:<12}")),
        a.yellow(&format!("★ {}", fmt_u64(repo.stargazers_count))),
        a.dim(&format!("⑂ {}", fmt_u64(repo.forks_count))),
        a.dim(pushed),
    );
    if !desc.is_empty() {
        println!("  {}", a.dim(desc));
    }
}

// ── Commit printer ────────────────────────────────────────────────────

fn print_commit(a: &Ansi, repo: &str, detail: &CommitDetail, branch: &str) {
    let short_sha = detail.sha.get(..7).unwrap_or(&detail.sha);
    let date = detail
        .commit
        .author
        .date
        .get(..10)
        .unwrap_or(&detail.commit.author.date);
    let msg = detail.commit.message.lines().next().unwrap_or("");
    let author = &detail.commit.author.name;

    println!(
        "  {}  {}  {}  {}  {}",
        a.yellow(short_sha),
        a.dim(date),
        a.cyan(repo),
        a.dim(&format!("({branch})")),
        a.bold(msg),
    );
    println!("  {}  {}", a.dim("              "), a.dim(author));

    let max_fname = detail
```

```rust
        .files
        .iter()
        .map(|f| f.filename.len())
        .max()
        .unwrap_or(0)
        .min(50);

    detail.files.iter().for_each(|f| {
        let status_label = match f.status.as_str() {
            "added" => a.green("added   "),
            "removed" | "deleted" => a.red("deleted "),
            "renamed" => a.magenta("renamed "),
            _ => a.dim("modified"),
        };
        println!(
            "  {}  {:<fname_w$}  {}  {}",
            a.dim("         "),
            f.filename,
            status_label,
            a.dim(&format!(
                "+{:<5} -{:<5}",
                fmt_u64(f.additions),
                fmt_u64(f.deletions)
            )),
            fname_w = max_fname,
        );
    });
    println!();
}

// ── Tree renderer ────────────────────────────────────────────────────────────

struct TreeNode {
    children: BTreeMap<String, TreeNode>,
    is_file: bool,
}

fn build_tree(paths: &[PathBuf]) -> TreeNode {
    let mut root = TreeNode {
        children: BTreeMap::new(),
        is_file: false,
    };
    for path in paths {
        let components: Vec<String> = path
            .components()
            .map(|c| c.as_os_str().to_string_lossy().into_owned())
            .collect();
        let total = components.len();
        let mut node = &mut root;
        for (i, name) in components.into_iter().enumerate() {
            node = node.children.entry(name).or_insert(TreeNode {
                children: BTreeMap::new(),
                is_file: i == total - 1,
            });
        }
    }
    root
}

fn print_tree(a: &Ansi, paths: &[PathBuf], root_name: &str) {
    let tree = build_tree(paths);
    println!("  {}/", a.bold(root_name));
    print_tree_children(a, &tree, "  ");
}

fn print_tree_children(a: &Ansi, node: &TreeNode, prefix: &str) {
    // Directories first (sorted), then files (sorted) — BTreeMap is already sorted.
    let dirs: Vec<(&String, &TreeNode)> =
        node.children.iter().filter(|(_, n)| !n.is_file).collect();
    let files: Vec<(&String, &TreeNode)> =
        node.children.iter().filter(|(_, n)| n.is_file).collect();

    let children: Vec<(&String, &TreeNode)> = dirs.into_iter().chain(files).collect();
    let last = children.len().saturating_sub(1);

    children.iter().enumerate().for_each(|(i, (name, child))| {
        let (connector, continuation) = if i == last {
            ("└── ", "    ")
```

```
        } else {
            ("├── ", "│   ")
        };
        if child.is_file {
            println!("{prefix}{connector}{name}");
        } else {
            println!("{prefix}{connector}{}/", a.bold(name));
            print_tree_children(a, child, &format!("{prefix}{continuation}"));
        }
    });
}

#[cfg(test)]
mod tests {
    use super::*;
    use tempfile::TempDir;

    // ── format_number ──────────────────────────────────────────────

    #[test]
    fn format_number_zero() {
        assert_eq!(format_number(0), "0");
    }

    #[test]
    fn format_number_small() {
        assert_eq!(format_number(42), "42");
        assert_eq!(format_number(999), "999");
    }

    #[test]
    fn format_number_thousands() {
        assert_eq!(format_number(1_000), "1,000");
        assert_eq!(format_number(12_345), "12,345");
    }

    #[test]
    fn format_number_millions() {
        assert_eq!(format_number(1_000_000), "1,000,000");
        assert_eq!(format_number(1_234_567), "1,234,567");
    }

    #[test]
    fn fmt_u64_rounds_trip() {
        assert_eq!(fmt_u64(0), "0");
        assert_eq!(fmt_u64(1_000), "1,000");
        assert_eq!(fmt_u64(1_000_000), "1,000,000");
    }

    // ── build_tree ─────────────────────────────────────────────────

    #[test]
    fn build_tree_empty() {
        let tree = build_tree(&[]);
        assert!(tree.children.is_empty());
        assert!(!tree.is_file);
    }

    #[test]
    fn build_tree_single_top_level_file() {
        let tree = build_tree(&[PathBuf::from("main.rs")]);
        assert_eq!(tree.children.len(), 1);
        let node = &tree.children["main.rs"];
        assert!(node.is_file);
        assert!(node.children.is_empty());
    }

    #[test]
    fn build_tree_nested_file() {
        let tree = build_tree(&[PathBuf::from("src/lib.rs")]);
        assert_eq!(tree.children.len(), 1);
        let src = &tree.children["src"];
        assert!(!src.is_file);
        assert_eq!(src.children.len(), 1);
        assert!(src.children["lib.rs"].is_file);
    }

    #[test]
```

```rust
    fn build_tree_dirs_and_files_separated() {
        let paths = vec![
            PathBuf::from("README.md"),
            PathBuf::from("src/main.rs"),
            PathBuf::from("src/lib.rs"),
        ];
        let tree = build_tree(&paths);
        assert_eq!(tree.children.len(), 2);
        assert!(!tree.children["src"].is_file);
        assert!(tree.children["README.md"].is_file);
        assert_eq!(tree.children["src"].children.len(), 2);
    }

    #[test]
    fn build_tree_deeply_nested() {
        let paths = vec![PathBuf::from("a/b/c/deep.rs")];
        let tree = build_tree(&paths);
        let a = &tree.children["a"];
        assert!(!a.is_file);
        let b = &a.children["b"];
        assert!(!b.is_file);
        let c = &b.children["c"];
        assert!(!c.is_file);
        assert!(c.children["deep.rs"].is_file);
    }

    #[test]
    fn build_tree_sorted_alphabetically() {
        let paths = vec![
            PathBuf::from("z.rs"),
            PathBuf::from("a.rs"),
            PathBuf::from("m.rs"),
        ];
        let tree = build_tree(&paths);
        let keys: Vec<&String> = tree.children.keys().collect();
        assert_eq!(keys, vec!["a.rs", "m.rs", "z.rs"]);
    }

    #[test]
    fn build_tree_duplicate_paths_idempotent() {
        let paths = vec![PathBuf::from("src/main.rs"), PathBuf::from("src/main.rs")];
        let tree = build_tree(&paths);
        assert_eq!(tree.children["src"].children.len(), 1);
    }

    // ── preview::repo ──────────────────────────────────────────────────
    // Verify the full preview pipeline returns Ok on the same inputs as the PDF
    // pipeline.  Stdout output is not captured — Ok(()) is the meaningful contract.

    async fn git(dir: &std::path::Path, args: &[&str]) -> anyhow::Result<()> {
        let p = dir
            .to_str()
            .ok_or_else(|| anyhow::anyhow!("non-utf8 path"))?;
        tokio::process::Command::new("git")
            .args(["-C", p])
            .args(args)
            .output()
            .await?;
        Ok(())
    }

    fn base_config(repo_path: std::path::PathBuf) -> crate::types::Config {
        crate::types::Config {
            repo_path,
            output_path: std::path::PathBuf::from("/tmp/unused.pdf"),
            include_patterns: vec![],
            exclude_patterns: vec![],
            theme: "InspiredGitHub".to_string(),
            font_size: 8.0,
            no_line_numbers: false,
            toc: true,
            file_tree: true,
            branch: None,
            commit: None,
            paper_size: crate::types::PaperSize::A4,
            landscape: false,
            remote_url: None,
        }
```

```
    }

    #[tokio::test]
    async fn preview_repo_git_repo() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        git(dir.path(), &["init", "-b", "main"]).await?;
        git(dir.path(), &["config", "user.email", "t@t.com"]).await?;
        git(dir.path(), &["config", "user.name", "T"]).await?;
        tokio::fs::write(dir.path().join("main.rs"), "fn main() {}\n").await?;
        git(dir.path(), &["add", "."]).await?;
        git(dir.path(), &["commit", "-m", "init"]).await?;

        crate::preview::repo(&base_config(dir.path().to_path_buf())).await
    }

    #[tokio::test]
    async fn preview_repo_plain_directory() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        tokio::fs::write(dir.path().join("hello.rs"), "fn main() {}\n").await?;

        crate::preview::repo(&base_config(dir.path().to_path_buf())).await
    }

    #[tokio::test]
    async fn preview_repo_empty_directory() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        crate::preview::repo(&base_config(dir.path().to_path_buf())).await
    }

    #[tokio::test]
    async fn preview_repo_single_file_plain() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        let file = dir.path().join("main.rs");
        tokio::fs::write(&file, "fn main() { println!(\"hi\"); }\n").await?;

        crate::preview::repo(&crate::types::Config {
            repo_path: file,
            ..base_config(dir.path().to_path_buf())
        })
        .await
    }

    #[tokio::test]
    async fn preview_repo_single_file_in_git_repo() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        git(dir.path(), &["init", "-b", "main"]).await?;
        git(dir.path(), &["config", "user.email", "t@t.com"]).await?;
        git(dir.path(), &["config", "user.name", "T"]).await?;
        tokio::fs::write(dir.path().join("lib.rs"), "pub fn f() {}\n").await?;
        git(dir.path(), &["add", "."]).await?;
        git(dir.path(), &["commit", "-m", "init"]).await?;

        crate::preview::repo(&crate::types::Config {
            repo_path: dir.path().join("lib.rs"),
            ..base_config(dir.path().to_path_buf())
        })
        .await
    }

    #[tokio::test]
    async fn preview_repo_no_file_tree() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        tokio::fs::write(dir.path().join("a.rs"), "fn a() {}\n").await?;

        crate::preview::repo(&crate::types::Config {
            file_tree: false,
            ..base_config(dir.path().to_path_buf())
        })
        .await
    }

    #[tokio::test]
    async fn preview_repo_include_filter() -> anyhow::Result<()> {
        let dir = TempDir::new()?;
        tokio::try_join!(
            tokio::fs::write(dir.path().join("main.rs"), "fn main() {}\n"),
            tokio::fs::write(dir.path().join("README.md"), "# hi\n"),
        )?;
```

```rust

        crate::preview::repo(&crate::types::Config {
            include_patterns: vec!["*.rs".to_string()],
            ..base_config(dir.path().to_path_buf())
        })
        .await
    }

    #[tokio::test]
    async fn preview_repo_nonexistent_path_errors() {
        assert!(
            crate::preview::repo(&base_config(std::path::PathBuf::from(
                "/nonexistent/preview/path"
            )))
            .await
            .is_err()
        );
    }
}
```

src/types.rs 239 LOC · 7.4 KB · 2026-03-14

```rust
use std::path::PathBuf;

/// Activity filter for the user report event feed.
#[derive(Debug, Clone, Copy, PartialEq, Eq, clap::ValueEnum)]
pub enum ActivityFilter {
    /// Show all event types (pushes, PRs, issues, stars, etc.)
    All,
    /// Show only push events (commits to repos)
    Commits,
}

/// Configuration for a `gitprint user` run.
#[allow(missing_docs)]
#[derive(Debug, Clone)]
pub struct UserReportConfig {
    pub username: String,
    pub output_path: PathBuf,
    pub paper_size: PaperSize,
    pub landscape: bool,
    /// Number of most-recently-pushed repos to include (0 = skip section).
    pub last_repos: usize,
    /// Number of recent commits with diffs to render (0 = skip diffs).
    pub last_commits: usize,
    /// Skip diff rendering entirely.
    pub no_diffs: bool,
    /// Font size used for diff/code blocks.
    pub font_size: f64,
    /// GitHub personal access token (`GITHUB_TOKEN` env var).
    pub github_token: Option<String>,
    /// Earliest date to include events from, in `YYYY-MM-DD` form (`None` = no lower bound).
    pub since: Option<String>,
    /// Latest date to include events from, in `YYYY-MM-DD` form (`None` = no upper bound).
    pub until: Option<String>,
    /// Which event types to include in the report.
    pub activity: ActivityFilter,
    /// Maximum number of events to show in the activity feed.
    pub events: usize,
}

/// Paper size for PDF output.
#[derive(Debug, Clone, Copy, clap::ValueEnum)]
pub enum PaperSize {
    /// ISO A4 (210 × 297 mm).
    A4,
    /// US Letter (215.9 × 279.4 mm).
    Letter,
    /// US Legal (215.9 × 355.6 mm).
    Legal,
}

/// Configuration for a gitprint run.
#[allow(missing_docs)]
#[derive(Debug, Clone)]
pub struct Config {
    pub repo_path: PathBuf,
    pub output_path: PathBuf,
    pub include_patterns: Vec<String>,
    pub exclude_patterns: Vec<String>,
    pub theme: String,
    pub font_size: f64,
    pub no_line_numbers: bool,
    pub toc: bool,
    pub file_tree: bool,
    pub branch: Option<String>,
    pub commit: Option<String>,
    pub paper_size: PaperSize,
    pub landscape: bool,
    /// Original remote URL when input was a remote repository, used for GitHub links.
    pub remote_url: Option<String>,
}

impl Config {
    #[cfg(test)]
    pub(crate) fn test_default() -> Self {
        Self {
            repo_path: PathBuf::from("."),
            output_path: PathBuf::from("/tmp/gitprint-test.pdf"),
```

```
            include_patterns: vec![],
            exclude_patterns: vec![],
            theme: "InspiredGitHub".to_string(),
            font_size: 8.0,
            no_line_numbers: false,
            toc: true,
            file_tree: true,
            branch: None,
            commit: None,
            paper_size: PaperSize::A4,
            landscape: false,
            remote_url: None,
        }
    }
}

/// Metadata extracted from a git repository.
#[allow(missing_docs)]
#[derive(Debug, Clone)]
pub struct RepoMetadata {
    pub name: String,
    pub branch: String,
    pub commit_hash: String,
    pub commit_hash_short: String,
    pub commit_date: String,
    pub commit_message: String,
    pub commit_author: String,
    /// Email address of the last committer.
    pub commit_author_email: String,
    pub file_count: usize,
    pub total_lines: usize,
    /// Filesystem owner of the input path (local paths only).
    pub fs_owner: Option<String>,
    /// Filesystem group of the input path (local paths only).
    pub fs_group: Option<String>,
    /// UTC timestamp when this PDF was generated.
    pub generated_at: String,
    /// Human-readable size of the git-tracked content (e.g. "4.2 MB").
    /// Computed from `git ls-tree -r -l`; empty for non-git paths.
    pub repo_size: String,
    /// Human-readable filesystem disk usage of the input path (e.g. "5.1 MB").
    /// Computed from `du -sh`; empty for remote repos.
    pub fs_size: String,
    /// Remote URL detected from git config (e.g. `git remote get-url origin`).
    /// Used to generate commit/author links even when `Config::remote_url` is None.
    pub detected_remote_url: Option<String>,
    /// Absolute filesystem path to the repo root (local repos only, `None` for remote clones).
    /// Used to generate `file://` links on the cover page.
    pub repo_absolute_path: Option<PathBuf>,
}

/// An RGB color value.
#[allow(missing_docs)]
#[derive(Debug, Clone, Copy)]
pub struct RgbColor {
    pub r: u8,
    pub g: u8,
    pub b: u8,
}

/// A single syntax-highlighted token with styling information.
#[allow(missing_docs)]
#[derive(Debug, Clone)]
pub struct HighlightedToken {
    pub text: String,
    pub color: RgbColor,
    pub bold: bool,
    pub italic: bool,
}

/// A line of syntax-highlighted tokens.
#[allow(missing_docs)]
#[derive(Debug, Clone)]
pub struct HighlightedLine {
    pub line_number: usize,
    pub tokens: Vec<HighlightedToken>,
}

```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_config_test_default() {
        let config = Config::test_default();
        assert_eq!(config.repo_path, PathBuf::from("."));
        assert_eq!(config.theme, "InspiredGitHub");
        assert_eq!(config.font_size, 8.0);
        assert!(config.toc);
        assert!(config.file_tree);
        assert!(!config.no_line_numbers);
        assert!(!config.landscape);
        assert!(config.branch.is_none());
        assert!(config.commit.is_none());
    }

    #[test]
    fn test_repo_metadata_clone() {
        let meta = RepoMetadata {
            name: "test".to_string(),
            branch: "main".to_string(),
            commit_hash: "abc123".to_string(),
            commit_hash_short: "abc1234".to_string(),
            commit_date: "2024-01-01".to_string(),
            commit_message: "init".to_string(),
            commit_author: "Alice".to_string(),
            commit_author_email: "alice@example.com".to_string(),
            file_count: 10,
            total_lines: 500,
            fs_owner: None,
            fs_group: None,
            generated_at: "2024-01-15 10:00:00 UTC".to_string(),
            repo_size: "1.2 MB".to_string(),
            fs_size: "1.5 MB".to_string(),
            detected_remote_url: None,
            repo_absolute_path: None,
        };
        let cloned = meta.clone();
        assert_eq!(cloned.name, "test");
        assert_eq!(cloned.file_count, 10);
    }

    #[test]
    fn test_rgb_color_copy() {
        let color = RgbColor {
            r: 255,
            g: 128,
            b: 0,
        };
        let copied = color;
        assert_eq!(copied.r, 255);
        assert_eq!(copied.g, 128);
        assert_eq!(copied.b, 0);
        // Original still usable (Copy trait)
        assert_eq!(color.r, 255);
    }

    #[test]
    fn test_highlighted_line_structure() {
        let line = HighlightedLine {
            line_number: 42,
            tokens: vec![
                HighlightedToken {
                    text: "fn".to_string(),
                    color: RgbColor { r: 0, g: 0, b: 255 },
                    bold: true,
                    italic: false,
                },
                HighlightedToken {
                    text: " main".to_string(),
                    color: RgbColor { r: 0, g: 0, b: 0 },
                    bold: false,
                    italic: false,
                },
            ],
        };
```

```
        assert_eq!(line.line_number, 42);
        assert_eq!(line.tokens.len(), 2);
        assert!(line.tokens[0].bold);
        assert!(!line.tokens[1].bold);
    }
}
```

src/user_report.rs 552 LOC · 18.8 KB · 2026-03-14

```rust
//! User report pipeline: fetch GitHub data in parallel, then render PDF.

use tokio::task::JoinSet;

use crate::github::{self, CommitDetail, GitHubEvent, GitHubRepo, GitHubUser};
use crate::pdf;
use crate::types::{ActivityFilter, UserReportConfig};

/// Pre-fetched GitHub data consumed by the PDF render phase.
///
/// Separating the fetch phase from the render phase keeps the render logic
/// testable without any network I/O.
pub(crate) struct UserReportData {
    pub user: GitHubUser,
    pub total_stars: u64,
    pub starred_repos: Vec<GitHubRepo>,
    pub active_repos: Vec<GitHubRepo>,
    pub pushed_repos: Vec<GitHubRepo>,
    pub events: Vec<GitHubEvent>,
    pub commit_msgs: std::collections::HashMap<String, String>,
    pub commit_details: Vec<(String, CommitDetail)>,
}

/// Fetches all GitHub data for the user report (Phases 1 & 2).
///
/// Separated from [`run`] so that [`crate::preview`] can reuse the same fetch
/// logic without triggering PDF rendering.
pub(crate) async fn fetch_data(config: &UserReportConfig) -> anyhow::Result<UserReportData> {
    let token = config.github_token.as_deref();
    let username = &config.username;

    // ── Phase 1: parallel API fetches ──────────────────────────────
    let (user_res, starred_res, active_res, pushed_res, events_res, search_commits_res) = tokio::join!(
        github::get_user(username, token),
        github::get_user_starred_repos(username, 5, token),
        github::get_user_repos(username, "updated", 5, token),
        github::get_user_repos(username, "pushed", config.last_repos, token),
        github::get_user_events(username, 100, token),
        async {
            if config.no_diffs || config.last_commits == 0 {
                Ok(vec![])
            } else {
                github::search_user_commits(username, config.last_commits, token).await
            }
        },
    );

    let user = user_res?;
    let starred_repos = starred_res.unwrap_or_default();

    let events = {
        let raw = coalesce_push_events(events_res.unwrap_or_default());
        let date_filtered = raw.into_iter().filter(|e| {
            let date = e.created_at.get(..10).unwrap_or(&e.created_at);
            config.since.as_deref().is_none_or(|s| date >= s)
                && config.until.as_deref().is_none_or(|u| date <= u)
        });
        match config.activity {
            ActivityFilter::All => date_filtered.collect::<Vec<_>>(),
            ActivityFilter::Commits => date_filtered
                .filter(|e| e.kind == "PushEvent")
                .collect::<Vec<_>>(),
        }
    };

    let push_event_repos: std::collections::HashSet<String> = events
        .iter()
        .filter(|e| e.kind == "PushEvent")
        .map(|e| e.repo.name.clone())
        .collect();
    let other_event_repos: std::collections::HashSet<String> = events
        .iter()
        .filter(|e| e.kind != "PushEvent")
        .map(|e| e.repo.name.clone())
        .collect();

    let pushed_repos: Vec<_> = pushed_res
```

```rust
        .unwrap_or_default()
        .into_iter()
        .filter(|r| {
            !r.fork && (push_event_repos.is_empty() || push_event_repos.contains(&r.full_name))
        })
        .collect();

    let pushed_names: std::collections::HashSet<&str> =
        pushed_repos.iter().map(|r| r.full_name.as_str()).collect();
    let active_repos: Vec<_> = active_res
        .unwrap_or_default()
        .into_iter()
        .filter(|r| {
            !r.fork
                && !pushed_names.contains(r.full_name.as_str())
                && (other_event_repos.is_empty() || other_event_repos.contains(&r.full_name))
        })
        .collect();

    let total_stars: u64 = starred_repos.iter().map(|r| r.stargazers_count).sum();

    // ── Phase 2: fetch commit details in parallel ──────────────────────
    let search_commits = match search_commits_res {
        Ok(commits) => commits,
        Err(e) if e.to_string().contains("rate limit") => return Err(e),
        Err(_) => vec![],
    };
    let commit_msgs: std::collections::HashMap<String, String> = search_commits
        .iter()
        .map(|(_, sha, msg)| (sha.clone(), msg.clone()))
        .collect();

    let commit_details: Vec<(String, CommitDetail)> = if !config.no_diffs && config.last_commits > 0
    {
        let shas: Vec<(String, String)> = search_commits
            .into_iter()
            .map(|(repo, sha, _)| (repo, sha))
            .collect();
        eprintln!("Fetching {} commit diff(s)...", shas.len());
        let mut set: JoinSet<anyhow::Result<(String, CommitDetail)>> = JoinSet::new();
        shas.into_iter().for_each(|(repo, sha)| {
            let tok = token.map(str::to_string);
            set.spawn(async move {
                github::get_commit_detail(&repo, &sha, tok.as_deref())
                    .await
                    .map(|cd| (repo, cd))
            });
        });
        let mut details: Vec<(String, CommitDetail)> = set
            .join_all()
            .await
            .into_iter()
            .filter_map(|r| match r {
                Ok(pair) => Some(Ok(pair)),
                Err(e) if e.to_string().contains("rate limit") => Some(Err(e)),
                Err(_) => None,
            })
            .collect::<anyhow::Result<Vec<_>>>()?;
        details.sort_unstable_by(|(_, a), (_, b)| b.commit.author.date.cmp(&a.commit.author.date));
        details
    } else {
        vec![]
    };

    Ok(UserReportData {
        user,
        total_stars,
        starred_repos,
        active_repos,
        pushed_repos,
        events,
        commit_msgs,
        commit_details,
    })
}

/// Runs the full user report pipeline and writes a PDF to `config.output_path`.
pub async fn run(config: &UserReportConfig) -> anyhow::Result<()> {
```

```rust
    let start = std::time::Instant::now();

    eprintln!("Fetching GitHub data for @{}...", config.username);
    let data = fetch_data(config).await?;

    eprintln!("Rendering PDF...");
    let (doc, total_pages) = render_to_doc(config, &data)?;
    pdf::save_pdf(&doc, &config.output_path).await?;

    let elapsed = elapsed_str(start.elapsed());
    let pdf_size = tokio::fs::metadata(&config.output_path)
        .await
        .map(|m| m.len())
        .unwrap_or(0);
    eprintln!(
        "{} — {} pages, {}, {}",
        config.output_path.display(),
        total_pages,
        format_size(pdf_size),
        elapsed,
    );
    Ok(())
}

/// Render the user report PDF from pre-fetched data.
///
/// Returns the assembled `PdfDocument` (ready to save) and the page count.
/// No network I/O is performed — all data must be supplied via `data`.
pub(crate) fn render_to_doc(
    config: &UserReportConfig,
    data: &UserReportData,
) -> anyhow::Result<(printpdf::PdfDocument, usize)> {
    let mut doc = printpdf::PdfDocument::new(&format!("{} — GitHub User Report", config.username));
    let fonts = pdf::fonts::load_fonts(&mut doc)?;
    let mut builder = pdf::create_user_builder(config, fonts);

    // Cover page
    pdf::user_cover::render(&mut builder, &data.user, data.total_stars);

    // Activity feed — capped to the requested display limit.
    let display_events = &data.events[..config.events.min(data.events.len())];
    pdf::user_activity::render(&mut builder, display_events, &data.commit_msgs);

    // Repository sections — pass events + fetched commit msgs for rich context
    render_repos_section(
        &mut builder,
        "Top Starred Repositories",
        &data.starred_repos,
        5,
        &data.events,
        &data.commit_msgs,
    );
    render_repos_section(
        &mut builder,
        "Repos You Were Active In",
        &data.active_repos,
        5,
        &data.events,
        &data.commit_msgs,
    );
    render_repos_section(
        &mut builder,
        "Repos User Pushed To",
        &data.pushed_repos,
        config.last_repos,
        &data.events,
        &data.commit_msgs,
    );

    // Commit diffs
    if !data.commit_details.is_empty() {
        // Build SHA → branch from push events so each diff header can show the branch.
        let sha_to_branch: std::collections::HashMap<&str, &str> = data
            .events
            .iter()
            .filter(|e| e.kind == "PushEvent")
            .filter_map(|e| {
                let sha = e.payload["head"].as_str()?;
```

```rust
                let branch = e.payload["ref"].as_str()?.trim_start_matches("refs/heads/");
                Some((sha, branch))
            })
            .collect();

        let bold = builder.font(true, false).clone();
        let black = printpdf::Color::Rgb(printpdf::Rgb::new(0.0, 0.0, 0.0, None));
        builder.write_centered("Recent Commits", &bold, printpdf::Pt(16.0), black);
        builder.vertical_space(12.0);
        data.commit_details.iter().for_each(|(repo, detail)| {
            let branch = sha_to_branch.get(detail.sha.as_str()).copied();
            pdf::diff::render_commit(&mut builder, detail, repo, branch, config.font_size as f32);
        });
    }

    let pages = builder.finish();
    let page_count = pages.len();
    doc.with_pages(pages);
    Ok((doc, page_count))
}

// ── Helpers ──────────────────────────────────────────────────────────────

/// Keep only the first PushEvent per (date, repo, branch) — GitHub emits one per push, so a busy
/// day can produce many identical-looking entries. Keeping the first (newest) is sufficient.
fn coalesce_push_events(events: Vec<GitHubEvent>) -> Vec<GitHubEvent> {
    let mut seen = std::collections::HashSet::new();
    events
        .into_iter()
        .filter(|event| {
            if event.kind != "PushEvent" {
                return true;
            }
            let date = event.created_at.get(..10).unwrap_or(&event.created_at);
            let branch = event.payload["ref"].as_str().unwrap_or("");
            seen.insert((
                date.to_string(),
                event.repo.name.clone(),
                branch.to_string(),
            ))
        })
        .collect()
}

fn render_repos_section(
    builder: &mut crate::pdf::layout::PageBuilder,
    title: &str,
    repos: &[GitHubRepo],
    limit: usize,
    events: &[GitHubEvent],
    commit_msgs: &std::collections::HashMap<String, String>,
) {
    if limit == 0 || repos.is_empty() {
        return;
    }
    let capped: Vec<_> = repos.iter().take(limit).cloned().collect();
    pdf::user_repos::render(builder, title, &capped, events, commit_msgs);
}

fn format_size(bytes: u64) -> String {
    if bytes < 1024 {
        format!("{bytes} B")
    } else if bytes < 1024 * 1024 {
        format!("{:.1} KB", bytes as f64 / 1024.0)
    } else {
        format!("{:.1} MB", bytes as f64 / (1024.0 * 1024.0))
    }
}

fn elapsed_str(d: std::time::Duration) -> String {
    if d.as_millis() < 1000 {
        format!("{}ms", d.as_millis())
    } else {
        format!("{:.1}s", d.as_secs_f64())
    }
}

#[cfg(test)]
```

```rust
mod tests {
    use super::*;
    use crate::github::{CommitAuthor, CommitFile, CommitInfo, EventRepo, GitHubUser};
    use crate::types::{ActivityFilter, PaperSize};

    fn make_push_event(repo: &str) -> GitHubEvent {
        GitHubEvent {
            kind: "PushEvent".to_string(),
            repo: EventRepo {
                name: repo.to_string(),
            },
            payload: serde_json::json!({ "ref": "refs/heads/main", "commits": [] }),
            created_at: "2024-03-01T12:00:00Z".to_string(),
        }
    }

    #[test]
    fn format_size_bytes() {
        assert_eq!(super::format_size(0), "0 B");
        assert_eq!(super::format_size(1023), "1023 B");
    }

    #[test]
    fn format_size_kilobytes() {
        assert_eq!(super::format_size(1024), "1.0 KB");
    }

    #[test]
    fn format_size_megabytes() {
        assert_eq!(super::format_size(1024 * 1024), "1.0 MB");
    }

    #[test]
    fn elapsed_str_milliseconds() {
        assert_eq!(
            super::elapsed_str(std::time::Duration::from_millis(42)),
            "42ms"
        );
        assert_eq!(
            super::elapsed_str(std::time::Duration::from_millis(999)),
            "999ms"
        );
    }

    #[test]
    fn elapsed_str_seconds() {
        assert_eq!(
            super::elapsed_str(std::time::Duration::from_millis(1500)),
            "1.5s"
        );
    }

    #[test]
    fn render_repos_section_empty_is_noop() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = crate::pdf::fonts::load_fonts(&mut doc).unwrap();
        let uc = mock_config(0);
        let mut builder = crate::pdf::create_user_builder(&uc, fonts);
        let page_before = builder.current_page();
        super::render_repos_section(
            &mut builder,
            "title",
            &[],
            5,
            &[],
            &std::collections::HashMap::new(),
        );
        assert_eq!(builder.current_page(), page_before);
    }

    #[test]
    fn render_repos_section_zero_limit_is_noop() {
        let mut doc = printpdf::PdfDocument::new("test");
        let fonts = crate::pdf::fonts::load_fonts(&mut doc).unwrap();
        let uc = mock_config(0);
        let mut builder = crate::pdf::create_user_builder(&uc, fonts);
        let page_before = builder.current_page();
        super::render_repos_section(
```

```
            &mut builder,
            "title",
            &[crate::github::GitHubRepo {
                name: "x".into(),
                full_name: "a/x".into(),
                html_url: "https://github.com/a/x".into(),
                description: None,
                language: None,
                stargazers_count: 0,
                forks_count: 0,
                open_issues_count: 0,
                size: 0,
                pushed_at: None,
                updated_at: None,
                created_at: None,
                fork: false,
            }],
            0,
            &[],
            &std::collections::HashMap::new(),
        );
        assert_eq!(builder.current_page(), page_before);
    }

    #[test]
    fn coalesce_keeps_first_push_per_day_branch() {
        let events = vec![
            make_push_event("alice/a"),
            make_push_event("alice/a"),
            make_push_event("alice/a"),
        ];
        let out = coalesce_push_events(events);
        assert_eq!(out.len(), 1);
    }

    #[test]
    fn coalesce_keeps_different_branches_separate() {
        let mut ev2 = make_push_event("alice/a");
        ev2.payload["ref"] = serde_json::json!("refs/heads/dev");
        let events = vec![make_push_event("alice/a"), ev2];
        assert_eq!(coalesce_push_events(events).len(), 2);
    }

    #[test]
    fn coalesce_preserves_non_push_events() {
        let events = vec![
            GitHubEvent {
                kind: "WatchEvent".to_string(),
                repo: EventRepo {
                    name: "alice/a".to_string(),
                },
                payload: serde_json::json!({}),
                created_at: "2024-03-01T00:00:00Z".to_string(),
            },
            make_push_event("alice/a"),
            make_push_event("alice/a"),
        ];
        let out = coalesce_push_events(events);
        assert_eq!(out.len(), 2);
        assert_eq!(out[0].kind, "WatchEvent");
    }

    // ── render_to_doc offline tests ──────────────────────────────────

    fn mock_user() -> GitHubUser {
        GitHubUser {
            login: "alice".to_string(),
            name: Some("Alice".to_string()),
            bio: None,
            location: None,
            company: None,
            blog: None,
            email: None,
            public_repos: 5,
            followers: 10,
            following: 5,
            created_at: "2020-01-01T00:00:00Z".to_string(),
            html_url: "https://github.com/alice".to_string(),
```

```
        }
    }

    fn mock_config(commits: usize) -> UserReportConfig {
        UserReportConfig {
            username: "alice".to_string(),
            output_path: "/tmp/test-commits.pdf".into(),
            paper_size: PaperSize::A4,
            landscape: false,
            last_repos: 0,
            last_commits: commits,
            no_diffs: false,
            font_size: 8.0,
            github_token: None,
            since: None,
            until: None,
            activity: ActivityFilter::All,
            events: 0,
        }
    }

    /// Generate a mock `(repo, CommitDetail)` with a large diff patch so it
    /// occupies several lines in the rendered PDF.
    fn mock_commit_detail(idx: usize) -> (String, CommitDetail) {
        let patch: String = (0..80).map(|i| format!("+added line {i}\n")).collect();
        (
            format!("alice/repo{idx}"),
            CommitDetail {
                sha: format!("{idx:040x}"),
                html_url: format!("https://github.com/alice/repo{idx}/commit/{idx:040x}"),
                commit: CommitInfo {
                    message: format!("commit #{idx}: add many lines"),
                    author: CommitAuthor {
                        name: "Alice".to_string(),
                        date: format!("2024-03-{:02}T12:00:00Z", idx % 28 + 1),
                    },
                },
                files: vec![CommitFile {
                    filename: format!("src/module{idx}.rs"),
                    status: "modified".to_string(),
                    additions: 80,
                    deletions: 0,
                    patch: Some(patch),
                }],
            },
        )
    }

    fn empty_report_data() -> UserReportData {
        UserReportData {
            user: mock_user(),
            total_stars: 0,
            starred_repos: vec![],
            active_repos: vec![],
            pushed_repos: vec![],
            events: vec![],
            commit_msgs: std::collections::HashMap::new(),
            commit_details: vec![],
        }
    }

    #[test]
    fn render_to_doc_no_commits_succeeds() {
        let (_, pages) = render_to_doc(&mock_config(0), &empty_report_data()).unwrap();
        assert!(pages > 0);
    }

    /// More commits with large diffs must produce more PDF pages than zero commits.
    /// This verifies the `--last-commits` flag actually drives the diff render path.
    #[test]
    fn more_commits_yields_more_pages() {
        let (_, pages_baseline) = render_to_doc(&mock_config(0), &empty_report_data()).unwrap();

        let data_with_commits = UserReportData {
            commit_details: (0..10).map(mock_commit_detail).collect(),
            ..empty_report_data()
        };
        let (_, pages_with_commits) = render_to_doc(&mock_config(10), &data_with_commits).unwrap();
```

```
546
547         assert!(
548             pages_with_commits > pages_baseline,
549             "expected more pages with commits ({pages_with_commits}) than without ({pages_baseline})"
550         );
551     }
552 }
```

tests/integration.rs

```rust
use std::path::{Path, PathBuf};

use tempfile::TempDir;

use gitprint::types::{Config, PaperSize};

async fn git_in(dir: &str, args: &[&str]) {
    let output = tokio::process::Command::new("git")
        .args(["-C", dir])
        .args(args)
        .output()
        .await
        .unwrap();
    assert!(
        output.status.success(),
        "git {:?} failed: {}",
        args,
        String::from_utf8_lossy(&output.stderr)
    );
}

async fn create_test_repo() -> TempDir {
    let dir = TempDir::new().unwrap();
    let p = dir.path().to_str().unwrap().to_string();

    git_in(&p, &["init", "-b", "main"]).await;

    // git config writes must be sequential (both modify .git/config, git's file lock
    // is advisory and concurrent writes fail). File writes are independent so they
    // run in parallel with the sequential git config block.
    tokio::join!(
        async {
            git_in(&p, &["config", "user.email", "test@test.com"]).await;
            git_in(&p, &["config", "user.name", "Test"]).await;
        },
        async {
            tokio::try_join!(
                tokio::fs::write(
                    dir.path().join("main.rs"),
                    "fn main() {\n    println!(\"hello\");\n}\n",
                ),
                tokio::fs::write(
                    dir.path().join("lib.rs"),
                    "pub fn add(a: i32, b: i32) -> i32 {\n    a + b\n}\n",
                ),
                tokio::fs::write(dir.path().join("README.md"), "# Test Repo\n"),
                tokio::fs::create_dir_all(dir.path().join("src")),
            )
            .unwrap();
            // src/ now exists; write util.rs after create_dir_all completes.
            tokio::fs::write(
                dir.path().join("src/util.rs"),
                "// utility\npub fn noop() {}\n",
            )
            .await
            .unwrap();
        },
    );

    git_in(&p, &["add", "."]).await;
    git_in(&p, &["commit", "-m", "initial commit"]).await;

    dir
}

fn test_config(repo_path: PathBuf, output_path: PathBuf) -> Config {
    Config {
        repo_path,
        output_path,
        include_patterns: vec![],
        exclude_patterns: vec![],
        theme: "InspiredGitHub".to_string(),
        font_size: 8.0,
        no_line_numbers: false,
        toc: true,
        file_tree: true,
        branch: None,
```

```
        commit: None,
        paper_size: PaperSize::A4,
        landscape: false,
        remote_url: None,
    }
}

// ── git module tests ─────────────────────────────────────────

#[tokio::test]
async fn git_verify_repo_valid() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let info = gitprint::git::verify_repo(repo.path()).await?;
    assert!(info.is_git);
    assert!(info.scope.is_none());
    assert!(info.single_file.is_none());
    Ok(())
}

#[tokio::test]
async fn git_verify_repo_subdir() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let info = gitprint::git::verify_repo(&repo.path().join("src")).await?;
    assert!(info.is_git);
    assert_eq!(info.scope, Some(PathBuf::from("src")));
    assert!(info.single_file.is_none());
    Ok(())
}

#[tokio::test]
async fn git_verify_repo_single_file_in_git() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let info = gitprint::git::verify_repo(&repo.path().join("main.rs")).await?;
    assert!(info.is_git);
    assert_eq!(info.single_file, Some(PathBuf::from("main.rs")));
    assert!(info.scope.is_none());
    Ok(())
}

#[tokio::test]
async fn git_verify_repo_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
    let dir = TempDir::new()?;
    let info = gitprint::git::verify_repo(dir.path()).await?;
    assert!(!info.is_git);
    assert!(info.single_file.is_none());
    Ok(())
}

#[tokio::test]
async fn git_verify_repo_plain_file() -> Result<(), Box<dyn std::error::Error>> {
    let dir = TempDir::new()?;
    tokio::fs::write(dir.path().join("hello.rs"), "fn main() {}")
        .await
        .unwrap();
    let info = gitprint::git::verify_repo(&dir.path().join("hello.rs")).await?;
    assert!(!info.is_git);
    assert_eq!(info.single_file, Some(PathBuf::from("hello.rs")));
    Ok(())
}

#[tokio::test]
async fn git_verify_repo_nonexistent_path() {
    assert!(
        gitprint::git::verify_repo(Path::new("/nonexistent/path"))
            .await
            .is_err()
    );
}

#[tokio::test]
async fn git_get_metadata() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    let metadata = gitprint::git::get_metadata(repo.path(), &config, true, None).await?;

    assert!(!metadata.name.is_empty());
    assert_eq!(metadata.branch, "main");
    assert_eq!(metadata.commit_hash.len(), 40);
```

```rust
    assert!(metadata.commit_hash.chars().all(|c| c.is_ascii_hexdigit()));
    assert_eq!(metadata.commit_hash_short.len(), 7);
    assert_eq!(metadata.commit_message, "initial commit");
    assert!(!metadata.commit_date.is_empty());
    Ok(())
}

#[tokio::test]
async fn git_get_metadata_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
    let dir = TempDir::new()?;
    let config = test_config(dir.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    let metadata = gitprint::git::get_metadata(dir.path(), &config, false, None).await?;

    assert!(!metadata.name.is_empty());
    assert!(metadata.branch.is_empty());
    assert!(metadata.commit_hash.is_empty());
    assert!(metadata.commit_date.is_empty());
    Ok(())
}

#[tokio::test]
async fn git_get_metadata_with_branch() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let mut config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    config.branch = Some("main".to_string());
    let metadata = gitprint::git::get_metadata(repo.path(), &config, true, None).await?;
    assert_eq!(metadata.branch, "main");
    Ok(())
}

#[tokio::test]
async fn git_list_tracked_files() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    let files = gitprint::git::list_tracked_files(repo.path(), &config, true, None).await?;

    assert!(files.contains(&PathBuf::from("main.rs")));
    assert!(files.contains(&PathBuf::from("lib.rs")));
    assert!(files.contains(&PathBuf::from("src/util.rs")));
    assert!(files.contains(&PathBuf::from("README.md")));
    assert_eq!(files.len(), 4);
    Ok(())
}

#[tokio::test]
async fn git_list_files_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
    let dir = TempDir::new()?;
    tokio::try_join!(
        tokio::fs::write(dir.path().join("hello.rs"), "fn main() {}"),
        tokio::fs::create_dir(dir.path().join("sub")),
    )
    .unwrap();
    tokio::fs::write(dir.path().join("sub/world.rs"), "pub fn world() {}")
        .await
        .unwrap();
    let config = test_config(dir.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    let files = gitprint::git::list_tracked_files(dir.path(), &config, false, None).await?;

    assert!(files.contains(&PathBuf::from("hello.rs")));
    assert!(files.contains(&PathBuf::from("sub/world.rs")));
    assert_eq!(files.len(), 2);
    Ok(())
}

#[tokio::test]
async fn git_read_file_content() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    let content =
        gitprint::git::read_file_content(repo.path(), Path::new("main.rs"), &config).await?;

    assert!(content.contains("fn main()"));
    assert!(content.contains("println!"));
    Ok(())
}

#[tokio::test]
async fn git_read_file_content_nonexistent() {
```

```rust
    let repo = create_test_repo().await;
    let config = test_config(repo.path().to_path_buf(), PathBuf::from("/tmp/test.pdf"));
    let result =
        gitprint::git::read_file_content(repo.path(), Path::new("nonexistent.rs"), &config).await;
    assert!(result.is_err());
}

// ── full pipeline tests ─────────────────────────────────────

#[tokio::test]
async fn full_pipeline() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let config = test_config(repo.path().to_path_buf(), output_path.clone());

    gitprint::run(&config).await?;

    assert!(output_path.exists());
    assert!(std::fs::metadata(&output_path)?.len() > 0);
    Ok(())
}

#[tokio::test]
async fn full_pipeline_with_include_filter() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.include_patterns = vec!["*.rs".to_string()];

    gitprint::run(&config).await?;

    assert!(output_path.exists());
    assert!(std::fs::metadata(&output_path)?.len() > 0);
    Ok(())
}

#[tokio::test]
async fn full_pipeline_with_exclude_filter() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.exclude_patterns = vec!["*.md".to_string()];

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}

#[tokio::test]
async fn full_pipeline_no_toc_no_tree() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.toc = false;
    config.file_tree = false;

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}

#[tokio::test]
async fn full_pipeline_no_line_numbers() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.no_line_numbers = true;

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}
```

```rust
#[tokio::test]
async fn full_pipeline_landscape() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.landscape = true;

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}

#[tokio::test]
async fn full_pipeline_letter_paper() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.paper_size = PaperSize::Letter;

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}

#[tokio::test]
async fn full_pipeline_subdir() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let config = test_config(repo.path().join("src"), output_path.clone());

    gitprint::run(&config).await?;

    assert!(output_path.exists());
    assert!(std::fs::metadata(&output_path)?.len() > 0);
    Ok(())
}

#[tokio::test]
async fn full_pipeline_single_file() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let config = test_config(repo.path().join("main.rs"), output_path.clone());

    gitprint::run(&config).await?;

    assert!(output_path.exists());
    assert!(std::fs::metadata(&output_path)?.len() > 0);
    Ok(())
}

#[tokio::test]
async fn full_pipeline_plain_directory() -> Result<(), Box<dyn std::error::Error>> {
    let dir = TempDir::new()?;
    tokio::try_join!(
        tokio::fs::write(dir.path().join("main.rs"), "fn main() {}\n"),
        tokio::fs::write(
            dir.path().join("lib.rs"),
            "pub fn add(a: i32, b: i32) -> i32 { a + b }\n",
        ),
    )
    .unwrap();
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let config = test_config(dir.path().to_path_buf(), output_path.clone());

    gitprint::run(&config).await?;

    assert!(output_path.exists());
    assert!(std::fs::metadata(&output_path)?.len() > 0);
    Ok(())
}

#[tokio::test]
async fn full_pipeline_nonexistent_repo() {
```

```rust
    let out_dir = TempDir::new().unwrap();
    let output_path = out_dir.path().join("output.pdf");
    let config = test_config(PathBuf::from("/nonexistent/repo"), output_path);

    assert!(gitprint::run(&config).await.is_err());
}

#[tokio::test]
async fn full_pipeline_invalid_theme() {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new().unwrap();
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path);
    config.theme = "NonExistentTheme".to_string();

    let err = gitprint::run(&config).await.unwrap_err();
    assert!(err.to_string().contains("NonExistentTheme"));
    assert!(err.to_string().contains("--list-themes"));
}

#[tokio::test]
async fn full_pipeline_include_excludes_everything() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.include_patterns = vec!["*.nonexistent".to_string()];

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}

#[tokio::test]
async fn full_pipeline_custom_font_size() -> Result<(), Box<dyn std::error::Error>> {
    let repo = create_test_repo().await;
    let out_dir = TempDir::new()?;
    let output_path = out_dir.path().join("output.pdf");
    let mut config = test_config(repo.path().to_path_buf(), output_path.clone());
    config.font_size = 12.0;

    gitprint::run(&config).await?;
    assert!(output_path.exists());
    Ok(())
}
```