DemoCreate: Declarative, Deterministic Generation of Narrated Demos of Software and Research Papers

Daniel Ari Friedman

2026-06-04

Contents

1 Abstract

Producing an audio-visual demonstration — a codebase tour, a website walkthrough, a terminal session, or a guided reading of a research paper — is conventionally a manual, capture-first activity: a human drives a screen recorder, narrates live, and re-records whenever the source changes. The resulting artifact is opaque, non-reproducible, and expensive to edit. DemoCreate (import name `democreate`) reframes demo production as a *declarative, deterministic* compilation problem (fig. 1). A demo is not a recording but a value: an ordered stream of typed actions over a virtual environment (editor, terminal, browser, camera) interleaved with chunked narration, expressed as a single `Demo` artifact that round-trips losslessly through JSON and YAML. That value compiles through a fixed pipeline — synthesize narration, measure its timing against the audio itself, lay out a timeline, compose frames, animate, encode, and content-verify — into a provenance-signed HD video. The declarative spine merges two threads of prior art: CodeVideo's event-sourced virtual-IDE model (CodeVideo 2024), in which content is a replayable log of typed actions, and VSpeak's chunk-and-trigger narration model (VSpeak 2024), in which on-screen events are anchored to spoken words. The result is a single validated schema with no I/O and no heavy dependencies.

Four design commitments distinguish the system. First, every heavy backend sits behind an abstract interface

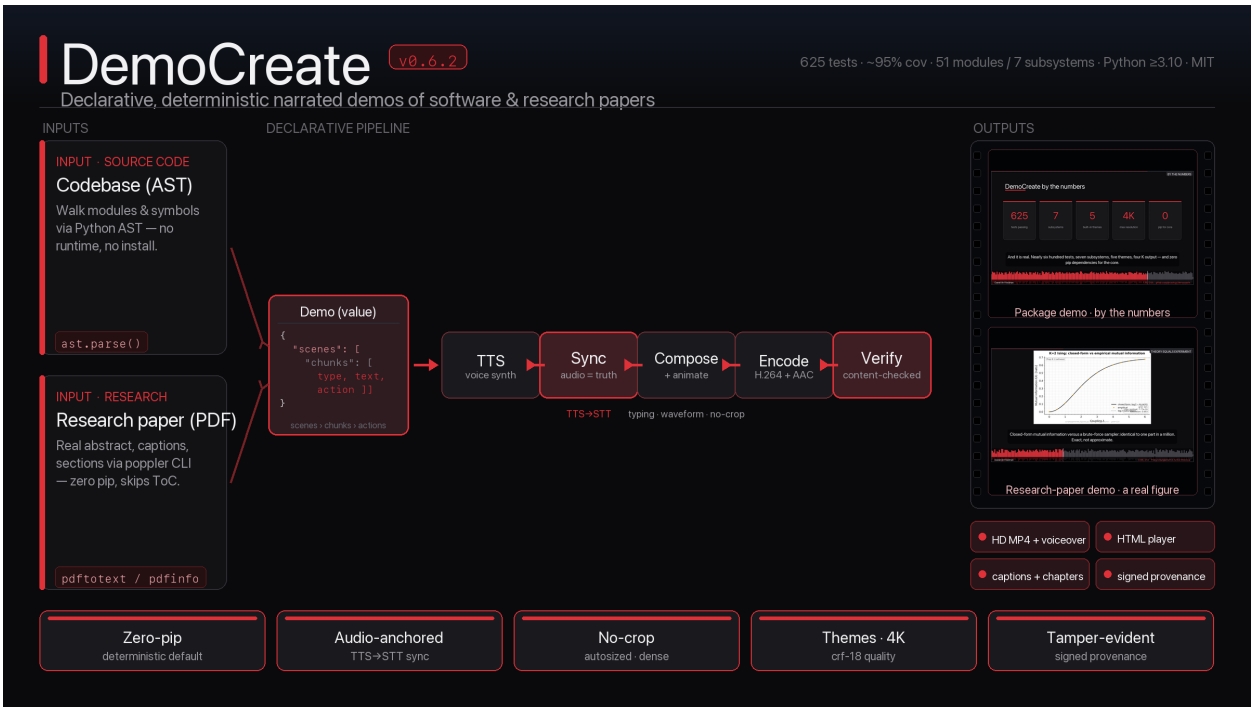

Figure 1: DemoCreate graphical abstract: a declarative demo of a codebase or a research paper compiles through a deterministic pipeline into a verified, provenance-signed HD video.

paired with a pure-Python deterministic default: text-to-speech behind a silent WAV writer (with a zero-pip real-voice `SystemTTSBackend` over macOS `say` / Linux `espeak`, and optional neural Kokoro (Hexgrad and Kokoro contributors 2025) / Chatterbox (Resemble AI 2025)), transcription (Whisper (Radford et al. 2023)) behind a heuristic word-distributor, screen and browser capture (`mss` (Schoentgen and python-mss contributors 2024), Playwright (Microsoft 2024)) behind a Pillow synthetic renderer, and video assembly behind a timed-frame animator that an `ffmpeg` (FFmpeg Developers 2024b) export turns into H.264. With only its light core dependencies installed, the package compiles a real, inspectable demo and *upgrades in fidelity, not capability*. Second, on-screen actions are synchronized to narration by a TTS→STT round-trip: narration audio is synthesized, transcribed back to word-level timestamps, and each action's `trigger_word` is fuzzy-matched against the transcript to receive an absolute millisecond timestamp — timing is *measured* from real audio rather than guessed, and each frame is held for its clip's measured duration. Third, the visual and audio composition is fully configurable: scaled TrueType fonts (Clark and Pillow contributors 2024), pygments syntax highlighting (Brandl et al. 2024), **five** preset themes (the default **noir** carries the design in black and white with a single red accent), bullet-list and stat-card presentation slides, character-by-character typing animation, an animated cursor with a click ripple, named aspect-ratio presets, resolution tiers up to 4K (3840×2160) at a near-visually-lossless CRF, a moving speech-waveform scrubber, a progress bar, scene crossfades, EBU R128 loudness normalization (European Broadcasting Union 2020), on-screen provenance overlay bars, MP4 container metadata tags, a signed steganographic provenance payload (Johnson and Jajodia 1998) hidden in lossless poster/bookend sidecars, and chapter / poster / GIF export are all threaded through a single `RenderConfig`. A *no-crop* layout treats every frame as a page: Ken Burns zoom is off by default because it crops content off the edges, background figures and PDF pages are fit *whole* (contain) with the caption below rather than over them, and code autosizes to the largest legible font that fits the frame and wraps rather than clipping. Fourth, DemoCreate demos *both software and research papers*: a `paper/` subsystem reads a PDF with the poppler utilities (The Poppler Developers 2024) (zero pip dependencies), extracts a title, the *real* abstract (skipping the table of contents), figure captions, sections, figures, and pages, and composes them — alongside a diagram of the paper's codebase — into a narrated demo.

The architecture is testable by construction: across 51 source modules in 7 subsystems, a real-filesystem suite

of 625 passing tests (3 skipped) holds a ≥90% coverage gate — achieving roughly 95% — with thin backend adapters excluded only where they cannot run without proprietary binaries, and a content verifier asserts that a rendered video carries real, non-silent, non-black streams. Its measured performance is recorded in a reproducible benchmark file: a 25.8 ms median build, 256.7 ms of render compute per second of output, and complete, monotonic synchronization timestamps. End to end, DemoCreate produced two content-verified 1920×1080 H.264 videos: a 128.4-second package demo in which the tool explains itself, and a 188.0-second demo of the *Policy Entanglement in Active Inference* paper — a 170-page PDF whose real ~1200-character abstract and section structure it extracted with the poppler CLI, alongside a diagram of the paper's 145-module codebase, narrated and normalized to a measured −15.5 dB mean volume (against a −16 LUFS target). We position the system honestly against CodeVideo, VSpeak, asciinema (Kulik and asciinema contributors 2024), termtosvg (Bedos 2020), code-video-generator (Wood and code-video-generator contributors 2023), Code2Video (Chen et al. 2025), Paper2Video (Zhu et al. 2025), and Manim (The Manim Community Developers 2024), and identify its distinctive contribution as the conjunction of a declarative deterministic spine, a backend-abstraction discipline that guarantees a working default, audio-anchored synchronization, and reproducible *research-paper* demos in a single dependency-light library.

2 Introduction

2.1 The Demo as a Disposable Artifact

A software demonstration — a guided tour of a codebase, a walkthrough of a web application, a recorded terminal session, or a narrated reading of the research paper behind the code — is one of the highest-leverage artifacts a project produces. It is also, in current practice, one of the least durable. The dominant workflow is *capture-first*: a human arranges a screen, starts a recorder, drives the interface by hand, narrates over a microphone, and edits the result in a video editor. The output is a pixel stream. It cannot be diffed, cannot be partially regenerated, and cannot be kept in sync with the software it depicts. When the source changes — a renamed function, a redesigned page, a new command-line flag, a revised figure — the only remedy is to re-record from scratch. The demo is born stale and dies disposable.

This fragility is not incidental; it is structural, and it mirrors a well-documented pattern elsewhere in computational work. Pimentel et al. found that only twenty-four percent of 1.4 million Jupyter notebooks on GitHub could be re-executed, and thirty-six percent produced different results when they ran (Pimentel et al. 2019) — a direct consequence of artifacts that encode *outcomes* rather than *reproducible descriptions*. A captured screencast is the audio-visual analogue of an un-re-executable notebook: it preserves the surface of a process while discarding the recipe. The remedy in both cases is the same in spirit — replace the captured outcome with a declarative, replayable description from which the outcome is *compiled* — and DemoCreate applies that remedy to the demo.

2.2 A Four-Stage Pipeline, and Where It Breaks

Generating an audio-visual demo, however it is built, decomposes into four stages. Naming them precisely is useful because each existing tool addresses some and ignores others, and because DemoCreate's contribution is to give all four a deterministic default. The four stages map directly onto the four columns of the system's architecture, shown in fig. 2 and developed in sec. 3.

The first stage is **capture**: producing the visual track — the editor showing a file, the terminal printing output, the browser loading a page, a paper figure filling the frame. Capture-first tools grab real pixels; this is faithful but non-reproducible and environment-dependent. asciinema (Kulik and asciinema contributors 2024) and termtosvg (Bedos 2020) take a more durable approach for terminals, recording a *timed event stream* (the asciicast format) that can be replayed deterministically — a model DemoCreate adopts and generalizes from terminals to editors, browsers, and slides.

The second stage is **narration**: producing the audio track. Modern open-weight TTS — Kokoro (Hexgrad and Kokoro contributors 2025), Chatterbox (Resemble AI 2025) — can synthesize technical narration without a human voice actor, but introducing a neural model makes the pipeline heavy, non-deterministic across

versions, and unrunnable in constrained environments such as continuous integration. DemoCreate's default avoids both extremes: a pure-Python silent backend for testing, and a *zero-pip* real-voice backend that drives the operating system's own speech synthesizer (`say` on macOS, `espeak` on Linux) for genuine narration with no Python dependency at all.

The third stage is **coordination**: aligning the audio and visual tracks in time, so that the on-screen file opens precisely when the narrator says "let us open the main module." This is the stage most tools either skip — leaving the author to nudge clips in a timeline by hand — or approximate by *estimating* narration duration in advance from word counts. Estimation is brittle: real speech does not match the estimate, and small drifts accumulate across a long demo until actions and words are visibly out of step. VSpeak's chunk-and-trigger model (VSpeak 2024) frames the problem correctly — anchor each action to a *spoken word* rather than to a guessed timestamp — but leaves open how the spoken word acquires a real time.

The fourth stage is **post-production**: assembling frames and audio into deliverables (a video, a captioned player, an interactive transcript). Here the toolchain is mature — `ffmpeg` (FFmpeg Developers 2024b) composes and encodes video; Manim (The Manim Community Developers 2024) animates — but each is a heavy binary dependency, and wiring them into a pipeline that *also* fails gracefully when they are absent is rarely attempted. DemoCreate's animator composes its own timed frames in pure Pillow — waveform, progress bar, crossfades, Ken Burns — and uses `ffmpeg` only for the final encode and audio loudness pass.

2.3 What DemoCreate Contributes

DemoCreate is a Python package (import name `democreate`) that treats all four stages as a single compilation from one declarative source, and applies that compilation to *both* software and research papers. Its contributions are developed in turn through the body of this manuscript.

First (sec. 3), a **declarative deterministic spine**. A demo is a `Demo` value — an ordered stream of typed `Action` objects mutating a virtual environment, interleaved with `Chunk` narration units — that validates its own structural invariants and round-trips losslessly through JSON and YAML. Rendering is a pure function of this value; editing means mutating the source and recompiling, never re-recording. The spine fuses CodeVideo's event-sourced action model (CodeVideo 2024; Fowler 2005) with VSpeak's chunk-and-trigger narration model (VSpeak 2024).

Second (sec. 3 and sec. 5), **backends behind interfaces, with deterministic defaults**. Every heavy capability — TTS, transcription, screen and browser capture, animation, video assembly, multi-language parsing, PDF ingestion — is an abstract interface whose default implementation is pure-Python or stdlib-only. The package therefore compiles a *real* demo with only its light core dependencies and upgrades in fidelity, not capability, when optional extras are installed.

Third (sec. 4), **TTS→STT synchronization**. Rather than estimating timing, DemoCreate synthesizes narration audio, transcribes it back to word-level timestamps, and anchors each action to its trigger word's *measured* start time. Even the deterministic default closes this loop: the silent TTS backend writes audio whose true duration encodes the estimated narration length, and the heuristic transcriber reads that real duration back, so the round-trip is exercised end-to-end with no neural model present.

Fourth (sec. 6 and sec. 7), **configurable composition and research-paper demos**. The look and motion of a render — five preset themes (the default **noir** is black and white with a single red accent), fonts, syntax highlighting, bullet and stat-card slides, the moving waveform, transitions, and loudness normalization — are controlled by a single serializable `RenderConfig`, and a *no-crop* layout law treats every frame as a page so that no figure, diagram, or line of code is ever lost to the frame edge. A dedicated `paper/` subsystem turns a PDF (plus its figures and codebase) into a narrated demo using the poppler command-line utilities (The Poppler Developers 2024) with no pip PDF dependency, recovering the paper's *real* abstract and section structure rather than boilerplate.

sec. 8 and sec. 9 address testability and reproducibility — the pure core is covered by a real-filesystem suite at a $\geq$90% gate, a content verifier asserts that rendered video is real and non-silent, and a `Demo` value re-renders identically on any machine — and sec. 10 compares the system honestly against its prior art.

3 Architecture

Three load-bearing ideas hold the architecture together: a declarative spine that keeps all content as data, a discipline of placing every heavy backend behind an abstract interface, and a guarantee that each interface ships a pure-Python deterministic default. The pipeline they compose is shown in fig. 2 as four left-to-right stages — declarative spine, narration, render, and export — each itself a column of interchangeable components. This section develops the three ideas in turn and then describes how the orchestrating pipeline composes them.

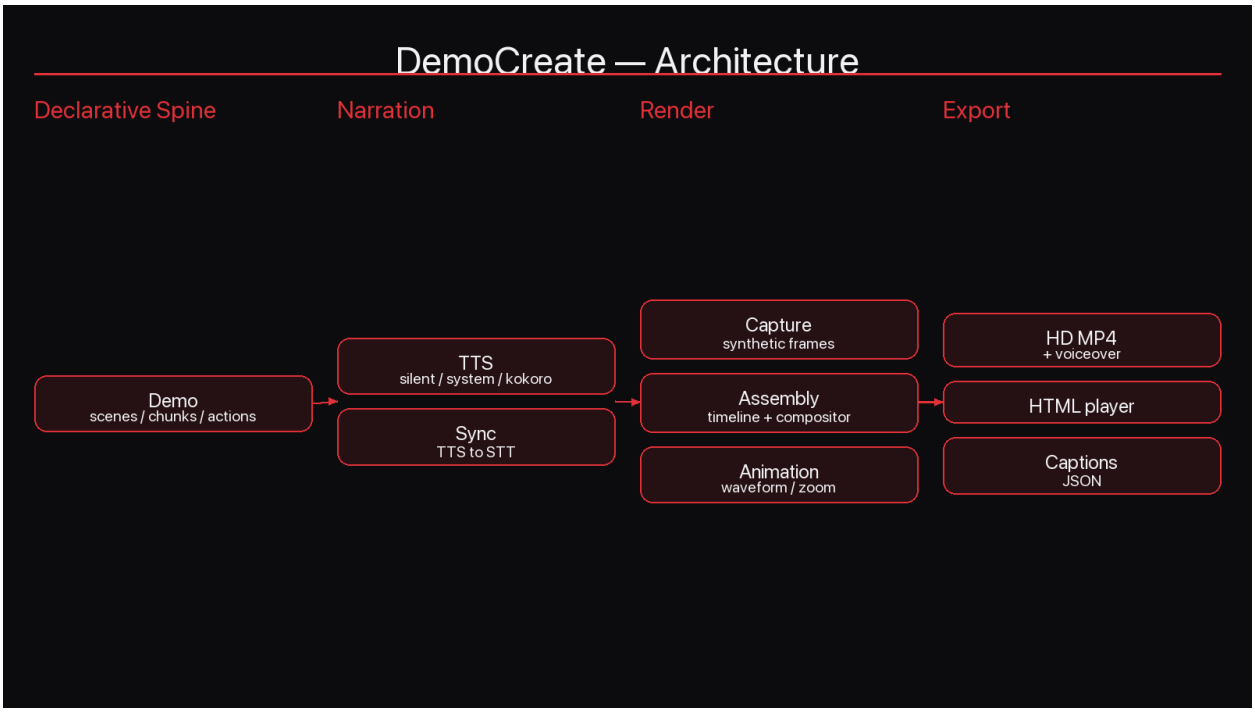

Figure 2: The canonical DemoCreate pipeline: a declarative `Demo` (scenes / chunks / actions) flows through narration (TTS plus TTS→STT sync), render (synthetic capture, timeline assembly, and waveform/zoom animation), and export (an HD MP4 with voiceover, a self-contained HTML player, and JSON captions). The figure is rendered by the package's own `democreate_architecture_image` diagram renderer.

3.1 The Declarative Spine

The single source of truth for a demo is the `Demo` value defined in `schema.py`. It is a tree of plain dataclasses with no I/O and no heavy dependencies, and rendering is a pure function of it. The supported edit workflow is to mutate the value and recompile — never to re-record.

A `Demo` owns an ordered list of `Scene` objects. Each `Scene` declares a `SceneKind` — one of `codebase`, `website`, `terminal`, or `slide` — which selects the capture and render strategy for that chapter, and holds an ordered list of `Chunk` objects. Each `Chunk` is a unit of narration: a block of `text` to be spoken, plus the `Action` objects that the narration triggers. Each `Action` carries a typed `ActionType`, a free-form `params` dictionary, and — crucially — an optional `trigger_word` naming a word in the parent chunk's narration to which the action is anchored. The action also carries `timestamp_ms` and `duration_ms` fields that are `None` until the synchronization stage fills them from real audio.

This structure deliberately fuses two prior-art models. The action stream is **event sourcing** in the sense of Fowler (Fowler 2005) and CodeVideo (CodeVideo 2024): content is an ordered, replayable log of typed mutations against a virtual environment, so the same log re-renders to any output format and any point in the demo is reconstructible by replaying the prefix. The chunk-and-trigger layering is VSpeak's narration model

(VSpeak 2024): narration is the organizing unit, and visual events are subordinated to spoken words rather than to wall-clock times. The `ActionType` enumeration spans the surfaces a demo needs — editor actions (`open_file`, `create_file`, `type_code`, `highlight_lines`, `close_file`), terminal actions (`run_command`, `print_output`), browser actions (`navigate`, `click`, `scroll`, `fill`), and camera/mouse actions (`move_mouse`, `zoom`, `pan`), plus the timing primitives `speak` and `wait`. The enum's string values are the stable on-disk representation and may not be renamed without a schema-version bump (`SCHEMA_VERSION`).

Three properties make the spine trustworthy. It **validates itself**: `Demo.validate()` returns a list of human-readable structural problems — empty title, non-positive frame geometry or frame rate, duplicate scene or chunk identifiers, actions with an invalid type — and never raises, leaving callers to choose strictness. It **round-trips losslessly**: `to_dict`/`from_dict`, `to_json`/`from_json`, and `to_yaml`/`from_yaml` are mutual inverses, and `Demo` equality is defined as dictionary equality, so `Demo.from_dict(d.to_dict()) == d` holds by construction. And it **estimates its own runtime** deterministically: in the absence of real audio, `estimated_duration_ms` derives a duration from narration word counts at a configurable words-per-minute pace (default 150), giving the timeline a sensible placeholder before any backend runs.

3.2 Backends Behind Interfaces

The second idea is a strict separation between *what* a stage does and *how* it is implemented. Every heavy capability is expressed as an abstract base class with a small, documented method surface, and concrete backends are interchangeable behind it. The pattern recurs identically across subsystems:

Subsystem	Abstract interface	Default (core)	Real-but-light	Heavy backends (extras)
Narration / TTS	`TTSBackend`	`SilentTTSBackend`	`SystemTTSBackend` (`say`/`espeak`)	`KokoroTTSBackend`, `ChatterboxTTSBackend`
Synchronization / STT	`Transcriber`	`HeuristicTranscriber`		`WhisperTranscriber`
Capture (screen)	`FrameSource`	`SyntheticRenderer`	—	`MssScreenCapture`
Assembly (compositing)	`Compositor`	`ManifestCompositor`	—	(video via `export/video.py` + `ffmpeg`)
Paper ingestion (PDF)	poppler CLI wrapper	`pdf.py` (`pdfinfo`/`pdftotext`/`pdftoppm`)	—	—

Backend selection is uniform: each subsystem exposes a `get_*` factory keyed on a name, where `"auto"` always resolves to the deterministic default. Heavy backends are *guarded* — their constructors probe for the optional dependency via `importlib.util.find_spec` (or `shutil.which` for a binary) and raise `BackendUnavailableError` (naming the missing package and the extra that provides it) when it is absent, rather than failing at import time. This is what lets the package be imported, tested, and run end-to-end in an environment that has none of the heavy binaries installed.

Two backends are worth singling out because they are *real* yet pull no pip dependencies. The `SystemTTSBackend` produces genuine spoken narration by shelling out to the operating system's built-in synthesizer (`say` on macOS, `espeak`/`espeak-ng` on Linux), transcoding the result to canonical 16-bit mono PCM via `ffmpeg` or `afconvert`, and *measuring* the resulting clip's duration. The `paper/pdf.py` wrapper reads, slices, and rasterizes PDFs through the poppler command-line utilities (The Poppler Developers 2024) that ship on most scientific workstations. Both deliver high-fidelity output while remaining installable with `pip install democreate` and nothing more.

3.3 Deterministic Defaults

The third idea is the one that gives the second its force: every default backend is not a stub but a *working, pure-Python implementation* that produces a real artifact. A stub would make the abstraction hollow — the package would import but do nothing useful without extras. A working default makes the abstraction honest: the package compiles a complete, inspectable demo with only its light core dependencies (`pyyaml`, `typer`, `rich`, `jinja2`, `pillow`).

The defaults are deliberately designed so that their outputs remain *meaningful* inputs to the rest of the pipeline. The silent TTS backend (`SilentTTSBackend`) does not return a dummy duration; it writes a valid 16-bit mono PCM WAV file of digital silence whose length is computed from the narration word count, so the file on disk has a true, measurable duration. The heuristic transcriber (`HeuristicTranscriber`) then reads that real duration back from the WAV and distributes the known words across it proportional to their character length — so the TTS→STT round-trip is genuinely closed even with no neural model present (sec. 4). The synthetic renderer (`SyntheticRenderer`) *draws* a clean, deterministic depiction of the editor, terminal, browser, or slide implied by each frame state using only Pillow (Clark and Pillow contributors 2024), rather than capturing real pixels — a "virtual desktop" in the spirit of CodeVideo. The manifest compositor (`ManifestCompositor`) writes a complete JSON render manifest plus one representative PNG per timeline entry. The consequence is that *fidelity*, not *capability*, is what an optional extra buys: a richer TTS swaps silence for synthesized speech; the video export swaps a frame manifest for an encoded H.264 file; nothing in the orchestration changes.

3.4 The Orchestrating Pipeline

The `Pipeline` class (`pipeline.py`) wires the subsystems together in a fixed, documented order, with each stage a pure function of the (mutated) demo plus a `Workspace`:

```
validate -> TTS -> TTS->STT sync -> timeline -> compose(frames + manifest)
         -> captions -> player + transcript + JSON
```

Concretely: the demo is validated (raising `SchemaValidationError` under `strict=True`, or logging warnings otherwise); `synthesize_demo` renders one audio clip per chunk; `sync_demo` transcribes that audio and assigns absolute timestamps to every chunk and action; `build_timeline` resolves a gap-free sequence of timeline entries; the configured compositor writes frames and a manifest; subtitle files are emitted in SRT and VTT; and an interactive HTML player, a Markdown transcript, and the serialized demo JSON are exported. The `render_video` path extends this by animating the per-chunk frames into a timed-frame sequence (sec. 6) and encoding them with `ffmpeg` against the assembled, normalized voiceover. The pipeline is constructed with any subset of backends overridden — `Pipeline(tts_backend=..., transcriber=..., compositor=..., config=...)` — and the `build_demo` convenience function constructs and runs a default pipeline in one call. The `Workspace` (`project_paths.py`) resolves and creates the output sub-directories (audio, frames, captions, manifests, web, demos), so no stage hard-codes a path. Because every stage consumes and produces the same plain values, the pipeline is itself just another pure function over the declarative spine.

4 Audio-Anchored Synchronization

The coordination stage — aligning on-screen actions with spoken narration in time — is the problem most demo tools solve poorly, and it is where DemoCreate's design is most opinionated. The governing principle is simple: **audio is the ground truth**. Each frame is held on screen for the *measured* duration of its narration clip, and every action is pinned to the *measured* onset of a spoken word. This section explains why timing must be measured rather than estimated, how the TTS→STT round-trip measures it, how trigger-word anchoring assigns each action an absolute time, how the gap-aware timeline keeps audio and video in lock-step, and how the heuristic and Whisper transcribers differ.

4.1 Estimate Versus Measure

The naive approach to coordination is to estimate narration duration in advance: count the words in a chunk, divide by a words-per-minute rate, and lay actions down on the resulting predicted timeline. DemoCreate does compute such estimates — `Chunk.estimated_duration_ms` exists, and the timeline builder falls back to it when no audio is present — but it treats them strictly as a *pre-render placeholder*, never as the authoritative timing. The reason is drift. Real synthesized speech does not match a uniform word rate: pauses, emphasis, numerals, and abbreviations all distort the mapping between word position and wall-clock time. An estimate that is off by a few hundred milliseconds per chunk is invisible at the start of a demo and badly desynchronized by the end, because the error accumulates over the action stream.

The correct discipline is to *measure* timing from the audio that will actually play. This requires closing a loop: synthesize the narration, then analyze the synthesized audio to recover when each word is spoken. VSpeak (VSpeak 2024) frames the goal — anchor each action to a spoken word — but the spoken word only acquires a usable timestamp once real audio exists and is analyzed. That analysis is speech-to-text with word-level alignment, and feeding TTS output back through STT is the round-trip at the center of this design.

4.2 The TTS→STT Round-Trip

The round-trip has two halves, each behind an interface (sec. 3). The first half, in `narration/tts.py`, synthesizes audio: `synthesize_demo` walks the demo's chunks and, for each, writes `<workspace.audio>/<chunk.id>.wav`, records the chunk's `audio_path`, and returns an `AudioClip` (carrying the file path, measured duration, sample rate, and source text) tagged with the chunk identifier. The duration on that clip is not an estimate — it is read back from the written file: the silent backend computes it from the frames it actually wrote, and the `SystemTTSBackend` measures it from the transcoded WAV via `measure_wav_duration_ms`, the audio-as-ground-truth primitive. The second half, in `narration/sync.py`, recovers word timings: a `Transcriber` turns an audio file plus its known text into a list of `WordTimestamp` objects, each a word with a millisecond `start_ms` and `end_ms`.

The pivotal design choice is that *the default backends close this loop with no neural model*. The silent TTS backend does not emit a zero-length or placeholder file; it writes a valid WAV whose true frame count — and therefore its true on-disk duration — encodes the estimated narration length. The heuristic transcriber then reads that real duration back via the stdlib `wave` module (`_wav_duration_ms`) and distributes the known words across the measured span. So even in a core-only environment, the system genuinely synthesizes audio, genuinely reads its real duration, and genuinely derives word timings from that measurement — the round-trip is exercised end-to-end, not simulated. When the `whisper` extra (or a real-voice TTS backend) is installed, the *same* round-trip carries real speech and real recognition with no change to the orchestration.

4.3 Trigger-Word Anchoring

With word timestamps in hand, `sync_demo` assigns absolute times across the whole demo. It indexes the audio clips by chunk identifier and walks the chunks in order, maintaining a running cumulative offset. Each chunk's `start_ms` is set to the cumulative sum of all preceding clip durations, so chunks are laid back-to-back on a single absolute timeline. Within a chunk, the transcriber produces the word timestamps local to that chunk's audio, and every action is resolved against them:

- If an action has a `trigger_word`, it is fuzzy-matched against the transcribed words (`_match_word`) using `difflib.get_close_matches` with a `0.6` similarity cutoff, after lowercasing and stripping surrounding punctuation. A match yields the spoken word's local `start_ms`, and the action's absolute `timestamp_ms` becomes `chunk.start_ms + word.start_ms`.
- If an action has no trigger word, or its trigger fails to match any spoken word, the action fires at the chunk's `start_ms` — a safe default that keeps it within its narration's window.
- Any action lacking an explicit `duration_ms` is given a sensible default play-out length.

Fuzzy matching, rather than exact equality, is deliberate: real transcription introduces casing, punctuation, and minor spelling differences between the authored trigger word and the recognized token, and a `0.6` cutoff tolerates those while still rejecting unrelated words. The companion function `absolute_word_timestamps`

applies the same per-chunk transcription and cumulative-offset shift to *every* word, yielding a single flat stream of absolute word timings suitable for word-level caption generation. The net effect is that on-screen events are pinned to the audio that will actually play: a file opens when the narrator says "open," because that word's start time was read from the rendered audio, not predicted from a word count.

4.4 The Gap-Aware Timeline

Audio-as-ground-truth only holds if the *video* timeline reserves time for exactly the silences the *audio* track contains. The voiceover is assembled by `concat_with_gaps` (sec. 6), which inserts a lead silence before the first clip, a configurable gap between consecutive clips for breathing room, and a trail silence after the last — all generated at the clips' own format with the stdlib `wave` module, so nothing resamples. The animator's `chunk_timing` lays out the per-chunk `(start_ms, end_ms)` windows using the *same* `lead_ms`, `gap_ms`, and `trail_ms`, so the frame timeline and the audio track share one clock. During lead and gap silences, `active_index_at` shows the *upcoming* chunk's frame, so the viewer sees the next scene a beat before its narration begins; after the final window the last frame holds. The result is that each base frame is displayed for precisely its clip's measured spoken duration, framed by the same silences the listener hears — the playhead and the waveform never drift from the words.

4.5 Heuristic Versus Whisper Transcription

DemoCreate ships two `Transcriber` implementations, selected by `get_transcriber` (where `"auto"` resolves to the heuristic default).

The `HeuristicTranscriber` is deterministic and stdlib-only. It requires the known narration text, reads the WAV's true duration, and lays the words end-to-end across that span, giving each word a slice proportional to its character length (longer words occupy proportionally more time, with a one-character floor to avoid zero-width slices), then snaps the final word's end to the measured duration. It performs no acoustic analysis — it has the *text* and the *true total duration* and interpolates between them — which is exactly why it is reproducible across machines and fast enough to run in continuous integration. Its limitation is honest: within a chunk, word boundaries are length-weighted approximations, not acoustic detections, so a trigger word's anchor is accurate to roughly its proportional position rather than to the precise onset of speech.

The `WhisperTranscriber` performs real speech recognition with word-level timestamps, guarded behind the `whisper` extra (Radford et al. 2023); its constructor raises `BackendUnavailableError` when the package is absent. Real Whisper output is the authoritative path for high-fidelity demos, and word-level alignment in the WhisperX style (Bain et al. 2023) provides genuinely acoustic word onsets, eliminating the heuristic's interpolation error. The architectural point is that swapping transcribers changes only the *accuracy* of the word timestamps, not the structure of synchronization: `sync_demo` consumes a `list[WordTimestamp]` regardless of which transcriber produced it, so the heuristic default and the Whisper upgrade are perfectly interchangeable behind the same anchoring logic.

5 Implementation

DemoCreate is organized as a single `democreate` package under `src/`: 51 source modules in total, a small set of shared top-level modules plus seven subsystem packages (`capture/`, `narration/`, `animation/`, `codebase/`, `assembly/`, `export/`, `paper/`). This section maps the module structure and describes each subsystem, emphasizing the deterministic default in each case. Two figures in this section show the rendering primitives directly: fig. 3 is a real synthetic editor frame, and fig. 4 is a real speech-waveform scrubber — both produced by calling the package's own public APIs.

5.1 Module Map

The top-level modules hold the shared, dependency-light primitives that the subsystems exchange:

Module	Responsibility
`schema.py`	The declarative spine: `Demo`, `Scene`, `Chunk`, `Action`, `ActionType`, `SceneKind`, `WordTimestamp` (sec. 3)
`config.py`	`Theme`, `AudioConfig`, `VideoConfig`, `RenderConfig`, and the `THEMES` presets (sec. 6)
`media.py`	Shared value types: `AudioClip` (a rendered narration file and its measured properties) and `FrameState` (a renderable snapshot of the virtual environment)
`pipeline.py`	The `Pipeline` orchestrator, `PipelineResult`, `build_demo`, and `render_video`
`cli.py`	The `democreate` command-line interface (Typer (Ramírez and Typer contributors 2024) + Rich (McGugan and Rich contributors 2024))
`project_paths.py`	The `Workspace` path resolver that creates and exposes all output sub-directories
`errors.py`	The exception hierarchy rooted at `DemoCreateError` (`SchemaValidationError`, `BackendUnavailableError`, `SyncError`, `RenderError`, …)
`_logging.py`	Self-contained structured logging and the `log_stage` context manager

5.2 `capture/` — The Visual Track

The capture subsystem produces the visual surface of each scene. Its centerpiece is the **synthetic renderer** in `capture/screen.py`: a `FrameSource` whose default `SyntheticRenderer` turns a `FrameState` into a Pillow image by *drawing* a clean, deterministic depiction of the editor, terminal, browser, or slide implied by that state — a virtual desktop in the CodeVideo lineage (CodeVideo 2024), reproducible across machines and requiring only the core `pillow` dependency (Clark and Pillow contributors 2024). Every metric is proportional to the frame height and all text uses real, scalable TrueType fonts resolved by `animation/fonts.py`, so titles and captions stay legible at HD. Code is colored with **pygments** (Brandl et al. 2024) when available — its lexer tokens mapped onto the active theme's syntax palette — falling back to a small keyword set otherwise. A frame may instead carry a `background_image` (a real browser screenshot, a generated diagram, or a paper figure), which the renderer fits *whole* into the content area — contain, never cropped (sec. 6) — placing the section pill in the chrome and the word-wrapped caption in its own band below the image, and reserving a bottom band for the animated waveform. fig. 3 is exactly such a frame: a line-numbered, pygments-highlighted editor showing real `schema.py` source with one highlighted line, a section pill, and a caption.

The real-pixel `MssScreenCapture` backend sits behind the `capture` extra (Schoentgen and python-mss contributors 2024) for grabbing genuine screen content, but is never required. `capture/terminal.py` models a terminal session as a stream of timed events serialized to the asciinema asciicast v2 format (Kulik and asciinema contributors 2024) — a header object followed by one `[time, kind, data]` array per line — so a list of `(command, output)` pairs becomes a deterministic recording and a sequence of renderable terminal frame states without launching a shell, echoing the durable-capture philosophy of asciinema and termtosvg (Bedos 2020). `capture/browser.py` drives `website` scenes, deterministically by default and via Playwright (Microsoft 2024) when the `browser` extra is present. `capture/replay.py` provides a pure event model for input record/replay, with guarded real backends over `pynput` (Palmér and pynput contributors 2024) and `pyautogui` (Sweigart and PyAutoGUI contributors 2024).

5.3 `narration/` — The Audio Track

The narration subsystem owns text-to-speech, script generation, and synchronization. `narration/tts.py` defines the `TTSBackend` interface, the default `SilentTTSBackend`, the zero-pip real-voice `SystemTTSBackend` (macOS `say` / Linux `espeak`), and the guarded `KokoroTTSBackend` (Hexgrad and Kokoro contributors 2025) and `ChatterboxTTSBackend` (Resemble AI 2025) (sec. 3). `narration/sync.py` defines the `Transcriber` interface, the deterministic `HeuristicTranscriber`, the guarded `WhisperTranscriber` (Radford et al. 2023),

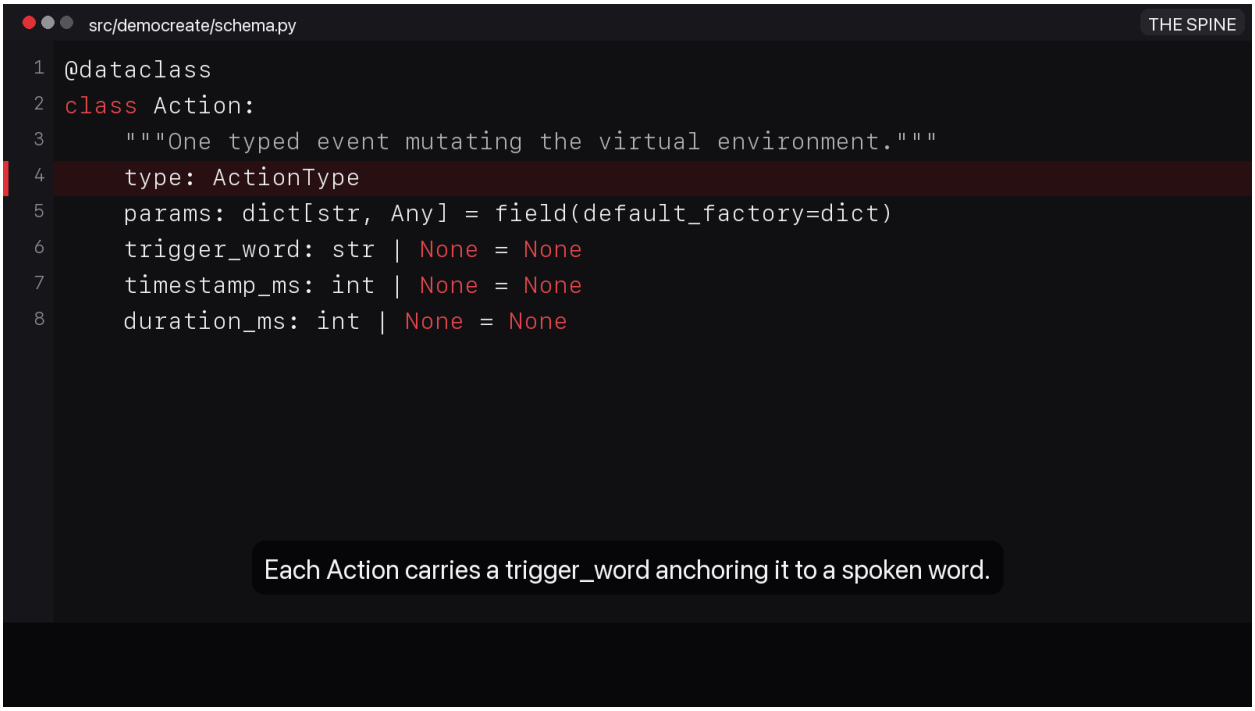

Figure 3: A synthetic `CODEBASE` editor frame rendered by `SyntheticRenderer`: scaled TrueType fonts, a line-number gutter, pygments syntax highlighting, an emphasized line, a section label pill, and a word-wrapped caption band — all drawn in pure Pillow with no real-pixel capture.

and the `sync_demo` and `absolute_word_timestamps` functions that close the TTS→STT round-trip (sec. 4). `narration/script.py` builds a declarative `Demo` from structured context — its `generate_codebase_demo` converts a list of module summaries (from the codebase subsystem) into scenes, chunks, trigger-bearing actions, and narration — so a demo can be *generated* programmatically rather than hand-authored.

5.4 `assembly/` — Timeline, Audio, and Animation

The assembly subsystem turns the demo into a rendered timeline and a moving video. `assembly/compositor.py` defines the pure `Timeline` data structure and `build_timeline`, which walks scenes and chunks to produce a gap-free, non-overlapping sequence of `TimelineEntry` objects — each pairing an absolute time window with a `FrameState` derived by replaying that chunk's actions (`_state_for_chunk`). It defines the `Compositor` interface and the default `ManifestCompositor` (which writes `render_manifest.json` and one PNG per entry, delegating frame drawing to the synthetic renderer).

`assembly/audio.py` post-processes the voiceover with pure-stdlib primitives plus guarded `ffmpeg` steps. `concat_with_gaps` concatenates the per-chunk WAVs in order, inserting lead, inter-chunk, and trail silences generated at the clips' own `(channels, sampwidth, framerate)` so nothing resamples; `measure_duration_ms` reads true durations from the WAV header. `normalize_audio` applies `ffmpeg`'s `loudnorm` filter — the EBU R128 (European Broadcasting Union 2020; FFmpeg Developers 2024a) integrated-loudness, true-peak, and loudness-range standard, defaulting to −16 LUFS — and `apply_fade` adds gentle in/out fades; both are guarded and raise `BackendUnavailableError` when `ffmpeg` is absent.

`assembly/animator.py` re-samples the one-frame-per-chunk output onto a fixed `animation_fps` and, for each instant, composites the active chunk's base frame, a moving **speech waveform** drawn into the reserved bottom band with a sweeping playhead, and a thin **progress bar** under the chrome; it crossfades across scene boundaries, re-renders flagged chunks through the **typing reveal** so code types in character-by-character, draws an **animated cursor** with a click ripple where a chunk supplies a `cursor_xy`, and carries

an optional **Ken Burns** zoom that is off by default so figures and pages stay un-cropped (sec. 6). The typing reveal is the most involved of these: for a flagged chunk the animator re-renders that chunk's `FrameState` at a `cursor_typed` count that grows with within-chunk time (capped so the chunk finishes typing after `typing_fraction` of its window), caching each distinct partial render so the cost is bounded by the number of *distinct* typed states rather than the frame count. The procedure is:

```
# Per output frame at time t_ms, for the active chunk idx (flagged for typing):
start, end = windows[idx]                       # the chunk's measured window
total = sum(len(line) for line in state.code_lines)
win   = max(1, end - start)
frac  = (t_ms - start) / win / typing_fraction   # 0..1 over the first 70%
typed = int(total * min(1.0, frac))             # characters revealed so far
key   = (idx, typed)
if key not in cache:                            # cache distinct partial renders
    s = copy(frame_states[idx]); s.cursor_typed = typed
    cache[key] = renderer.render(s, size)       # pygments re-highlights the partial source
frame = cache[key]                              # then overlay waveform, progress, cursor
```

The waveform is computed by `animation/waveform.py`, whose `compute_envelope` reduces a 16-bit PCM WAV to normalized RMS amplitude buckets (the only disk-touching step) and whose `draw_waveform` paints mirrored played/unplayed bars; fig. 4 shows the resulting scrubber at 55% progress. `animation/diagram.py` renders the architecture diagrams (such as fig. 2); `assembly/captions.py` emits SRT, VTT, and ASS subtitles purely from the demo, including a word-level path driven by `absolute_word_timestamps`; and `assembly/effects.py` provides pure Pillow image effects.

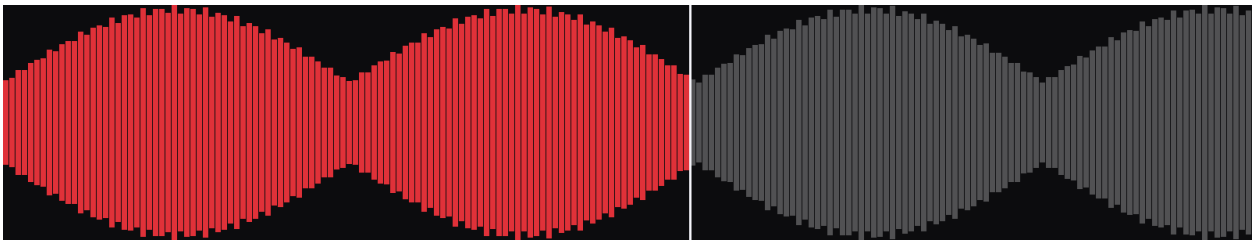

Figure 4: A speech-waveform scrubber rendered by `render_waveform_strip` from a real WAV envelope at 55% progress: mirrored amplitude bars with the played portion lit (left) and the unplayed portion dimmed (right), and a thin playhead at the boundary. The animator draws this band over every frame, locked to the measured audio.

5.5 `animation/`, `codebase/`, `export/`, and `paper/`

The remaining subsystems round out the build. `animation/zoom.py` and `animation/highlights.py` compute cursor-following zoom/pan and code emphasis purely; `animation/manim_scenes.py` specifies Manim (The Manim Community Developers 2024) code-walkthrough scenes, in the manner of code-video-generator (Wood and code-video-generator contributors 2023) and Code2Video (Chen et al. 2025), behind the `animation` extra. The codebase subsystem summarizes source for tours: `codebase/walker.py` extracts a `ModuleSummary` — docstring, top-level functions, classes and methods, imports, line count — using only the stdlib `ast` module, with `tree-sitter` (Brunsfeld and Tree-sitter contributors 2024) as the optional multi-language upgrade; `codebase/ast_viz.py` and `codebase/dependency.py` render summaries and import graphs.

The export subsystem produces deliverables. `export/formats.py` serializes a demo to JSON and a Markdown transcript; `export/interactive.py` renders a self-contained, dependency-free HTML player via a Jinja2 template, embedding the resolved timeline so the demo can be scrubbed in a browser with no server; `export/video.py` encodes the animated frames against the assembled, normalized voiceover into an H.264 MP4 with `ffmpeg` (the optional `video` extra swaps in MoviePy (Zulko and MoviePy contributors 2024) for

scripted compositing); and `export/verify.py` content-asserts the result (sec. 8). The `paper/` subsystem — `pdf.py`, `extract.py`, and `script.py` — turns a research paper into a narrated demo and is the subject of sec. 7. As elsewhere, the core build yields a genuine, shareable deliverable — the interactive HTML player and the transcript — while the heavy backends add the encoded video.

6 Composition and Configurability

A demo's content is fixed by its `Demo` value, but its *look*, *sound*, and *motion* are not — they are governed by a single `RenderConfig` (sec. 3) that is plain, serializable data with sensible defaults that reproduce the package's out-of-the-box appearance. Omit the config and nothing changes; supply one and a render can be reproduced from one file (`democreate render demo.json --config my_theme.yaml`) or selected by preset (`--theme paper`). A `RenderConfig` bundles four sections — `Theme`, `AudioConfig`, `VideoConfig`, and a `MetadataConfig` (sec. 11) — and the most accessible way to reach all of them is `democreate config out.yaml`, which writes a *fully-commented* default YAML (`RenderConfig.commented_yaml`) where every commonly-tuned knob — resolution and quality, motion, audio, and the provenance fields — carries an inline comment, ready to edit and pass back via `--config`. This section describes the three composition axes the config controls — visual look, motion, and audio — and how each is threaded through the renderer, animator, and audio assembler; the provenance axis is the subject of sec. 11.

6.1 Themes, Fonts, and Syntax Highlighting

The `Theme` dataclass holds the visual look as plain data: roughly two dozen RGB surface, text, accent, and syntax colors, plus six *font ratios* expressed as fractions of the frame height (title, subtitle, code, terminal, caption, and section). Because the ratios are fractions rather than pixel sizes, text stays proportional at any resolution — `scaled_font` in `animation/fonts.py` multiplies a ratio by the frame height and resolves a real system TrueType face (with a bitmap fallback). As of v0.6.2 the ratios were enlarged across *every* theme for legibility on video: a title at ratio `0.094` is a comfortable ~101 px on a 1080-line frame, captions at `0.038`, and the code autosizer's ceiling at `0.034` of the frame height, all scaling cleanly to any geometry. Code coloring is delegated to pygments (Brandl et al. 2024): the renderer lexes the source, maps each token class onto the theme's `syn_*` palette (keywords, strings, comments, numbers, names), and draws the colored spans, falling back to a flat keyword set only if pygments is unavailable.

Five presets ship in `THEMES`. The default is **noir**: near-black `(12,12,14)` surfaces, bright-white `(242,242,244)` text, and a *single* refined red `(224,49,57)` as the only chroma — spent sparingly on the played waveform, the cursor, code keywords, the section pill / heading rule, the line-highlight bar, and the top-edge progress line. Its code syntax is monochrome-with-red (keywords red, names white, strings gray, comments dim), so the eye reads structure rather than a rainbow. Alongside noir ship the classic `dark` (slate editor, blue accent), a high-contrast `light`, a deep-blue `midnight`, and an academic `paper` look with warm paper-white slides intended for research-paper demos (sec. 7). fig. 5 renders the *same* editor frame under all five, making the point that a theme change is a data change, not a code change — the layout, fonts, and highlighting are identical; only the palette differs. Partial themes are supported: `RenderConfig.from_dict` merges a user's overrides onto a named base, so a config can recolor a single accent while inheriting everything else.

Beyond code frames, the renderer composes title cards and figure backgrounds. A `SLIDE` frame draws a large centered title with an accent underline and an optional subtitle, as in fig. 6; when a `background_image` is set, that image is fit *whole* into the content area — never cropped — and the section pill and caption are placed *outside* it, the no-crop composition used for paper figures and rendered PDF pages (sec. 7) developed below.

6.2 Presentation Slides: Bullets and Stat Cards

A demo is not only code and figures; it also needs slides that *summarize*. The `SLIDE` frame therefore supports two presentation layouts beyond the title card, both driven by plain data on the `FrameState`. A **bullet slide** (`FrameState.bullets`) renders a titled list of short claims, each prefixed with an accent marker and sized

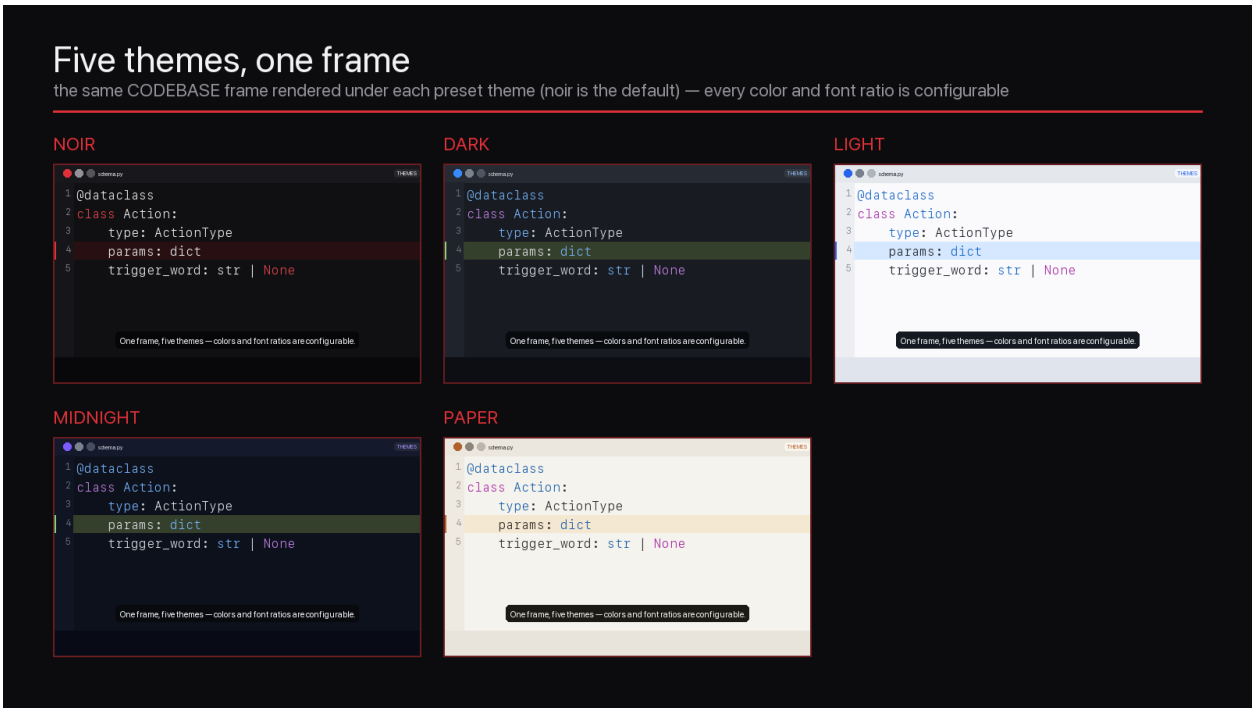

Figure 5: One synthetic editor frame rendered under all five preset themes — `noir` (the default), `dark`, `light`, `midnight`, and `paper`. Layout, fonts, pygments highlighting, the section pill, and the caption are identical across cells; only the `Theme` palette changes, illustrating that the look is configurable data rather than code. Noir carries the design in black and white with a single red accent.

Figure 6: A `SLIDE` title card rendered by `SyntheticRenderer`: a large TrueType title sized by the theme's `title_ratio`, an accent underline, a subtitle in the dim color, and a section pill in the top chrome. The same slide renderer composes intro, outro, and section-divider frames.

by the theme's font ratios, used for the "what it is" and "what to take away" beats of a tour — e.g. *"A demo is a value, not a recording."* A **stat-card slide** (`FrameState.stats`) renders a row of large number/label pairs for a by-the-numbers beat — e.g. *625 tests · 7 subsystems · 5 themes · 4K · 0 pip.* Both are pure `Theme`-colored compositions with no new dependency: the numbers and labels are authored content, the layout and palette are configuration, so a stat row recolors with a theme change exactly as a code frame does. The package's own showcase (sec. 8) uses both — bullet slides for its narrative beats and a stat card for its closing figures — so the manuscript's claims about the system are shown on screen as the system renders them.

6.3 The No-Crop Layout: Every Frame Is a Page

The defining composition discipline is *losing no content to the frame edge.* A demo of code or a research paper is information-dense — a figure, a diagram, a screenshot, a block of source all carry detail that a crop silently discards — so the renderer treats each frame as a *page* whose whole content must remain visible, and three mechanisms enforce that.

First, **Ken Burns zoom is off by default**. A slow center zoom looks cinematic on a photograph but is destructive on a figure: pushing in past 1.0 pushes the figure's own edges, axis labels, and corner annotations off-screen. The `ken_burns` flag therefore defaults to `False` (with `ken_burns_zoom` capped at a gentle `1.06` when a creator does opt in), so a still figure or PDF page is shown *complete* rather than cut. Second, **background images fit by contain, not cover**. `SyntheticRenderer._draw_background` scales an image by the *smaller* of the width and height ratios so the entire figure, diagram, or screenshot lands inside the content area — centered on a theme-colored matte with a thin border — where the old cover-crop scaled by the *larger* ratio and sheared off whatever overflowed. The caption then sits in its own band *below* the image rather than overlaying it, so neither the picture nor its label is ever occluded. Third, **code autosizes and wraps rather than clipping**. `_autosize_code` picks the largest legible monospace font for which both the longest line fits the frame width and every line fits the available height; a hard legibility floor (~18 px at 1080p) stops the font from shrinking indefinitely, and `_wrap_code_rows` flows any still-too-long line onto continuation rows with a hanging indent — so a long line is *wrapped*, never truncated at the margin. The net effect is that whatever the demo names, the viewer sees in full.

6.4 Motion: Waveform, Progress, Transitions, and Ken Burns

The `VideoConfig` controls geometry and motion. By default the animator (sec. 5) draws a moving speech waveform (fig. 4) and a thin progress line over every frame, both keyed to the measured audio timeline so they never drift from the spoken words. The progress line obeys a *no-overlap band law* that complements the no-crop discipline above: it is pinned to the **absolute top edge** of the frame (`y=0`, ~6 px tall) so it never overlaps the picture. It previously sat ~58 px down — just below the window chrome — where it clipped the top of full-frame figures and diagrams; moving it to the very top edge gives a clean band stack from top to bottom (progress line → content → caption → waveform), with content never sharing a row with chrome. In the default noir theme the line is drawn in the single red accent. `transitions` enables a crossfade across scene boundaries: when the active chunk's scene differs from the previous chunk's and the elapsed within-chunk time is under `transition_ms`, the animator alpha-blends the previous scene's frame into the current one. `ken_burns` would apply a slow center zoom (up to `ken_burns_zoom`, default `1.06`) to flagged slide and background frames, but it is *off by default* for the no-crop reason above — a still figure is shown whole rather than zoomed into. Every motion feature is independently toggleable, and the `AnimationConfig.from_video` constructor derives the animator's settings from the `VideoConfig` (and pulls the waveform colors from the `Theme` and the lead/gap/trail silences from the `AudioConfig`), so the three config sections stay consistent with one another.

6.5 Typing Reveal and the Animated Cursor

Two motion features give code scenes a sense of authorship rather than a static dump. The **typing reveal** types code in character-by-character. When a chunk is flagged for typing (a `type_code` or `create_file` action), the animator re-renders that chunk's `FrameState` at a progressively larger `cursor_typed` count for

each output frame, so the editor fills in one character at a time — and because the re-render runs through the same `SyntheticRenderer`, pygments re-highlights the partial source as it appears, with a block caret at the typing head. The reveal is paced by `typing_fraction` (default `0.7`): the chunk spends the first 70% of its measured window typing, then holds the completed frame while the narration finishes, so the words and the keystrokes land together. fig. 7 makes the mechanism visible: the *same* editor frame rendered at 25%, 55%, and 100% of `cursor_typed`, exactly the snapshots the animator composites between.

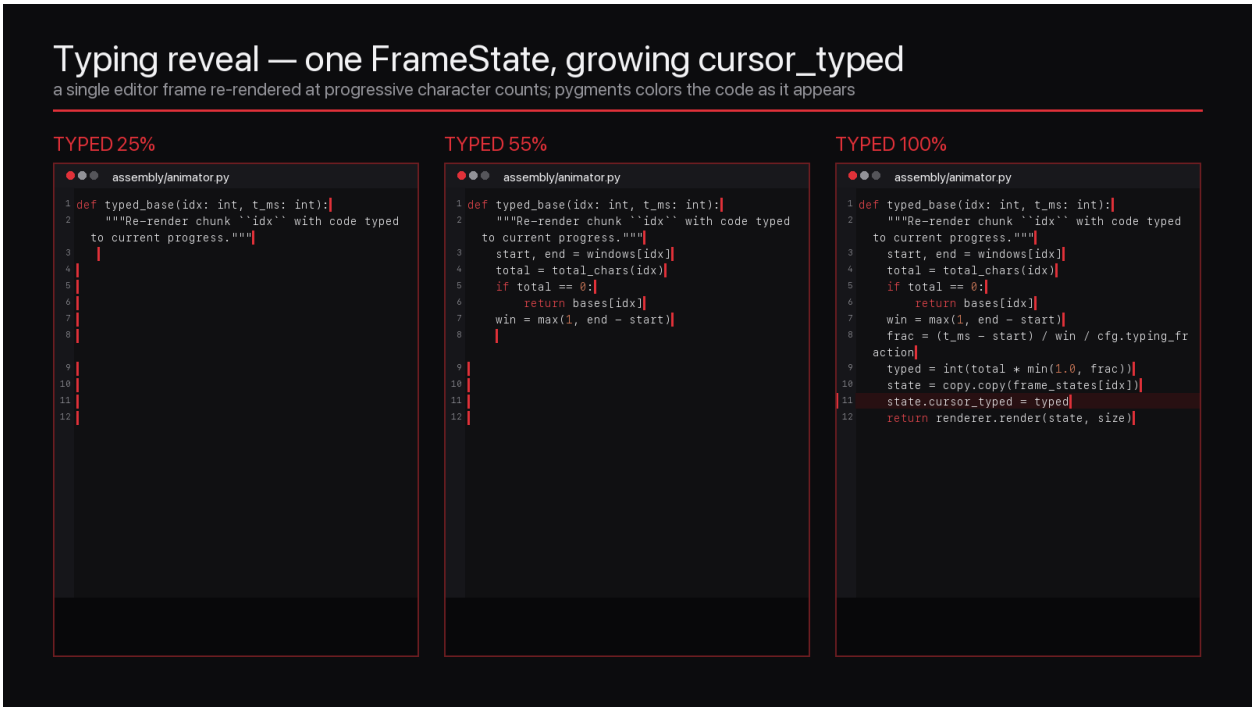

Figure 7: One `FrameState` re-rendered at `cursor_typed` of 25%, 55%, and 100% (left to right). The animator emits the intermediate frames between these snapshots, so code types in character-by-character with pygments re-highlighting the partial source and a caret at the typing head. Layout, theme, and gutter are identical across panels; what changes is how much of the source has been "typed", the caret position, and — once typing reaches it — the highlighted line.

The **animated cursor** complements typing. When a chunk supplies a `cursor_xy` position, the animator draws an arrow cursor at that point via `_draw_cursor`, and for the first 600 ms of the chunk paints an expanding, fading **click ripple** around it — a soft visual confirmation that an on-screen action occurred where the narration says it did. Like every other motion feature it is gated by a config flag (`cursor`, default on) and scales with the frame height, so it reads correctly at any aspect ratio.

6.6 Aspect-Ratio Presets and Resolution Tiers

Because every font size and layout metric in the renderer is a fraction of the frame height, a render retargets to a new geometry without any layout rework — the same `Demo` recomposes cleanly at any shape *or scale*. Named **aspect presets** live in `ASPECTS`: `16:9` (1920×1080, the default), `9:16` (1080×1920, vertical short-form), `1:1` (1080×1080), `4:3` (1440×1080), and `4:5` (1080×1350). `RenderConfig.set_aspect` sets `video.width`/`video.height` from a preset, and both `render` and `paper` accept `--aspect`, so producing a square or vertical cut of an existing demo is a one-flag change rather than a re-author.

Named 16:9 **resolution tiers** live in `RESOLUTIONS` — `720p` (1280×720), `1080p` (1920×1080), `1440p` (2560×1440), and `2160p/4k` (3840×2160) — selectable via `RenderConfig.set_resolution` or `render --resolution`. Because the same height-relative scaling governs every glyph and box, a higher tier is *genuinely* higher resolution rather than an upscale: text and chrome are re-rasterized at the larger size, not

stretched. Encode quality is governed by two `VideoConfig` knobs passed straight through to `ffmpeg` by `encode_frame_sequence`/`assemble_video`: `crf` (default **18**, near-visually-lossless — meaningfully below x264's ~23 default) and `preset` (the speed/compression trade-off). So the default render is crisp HD; opting up to 4K and tuning the CRF are one-flag (or one-field) changes that the rest of the pipeline (sec. 11) treats as ordinary configuration.

6.7 On-Screen Metadata Bars

The renderer can pin broadcast-style provenance onto the frame itself. `export/overlay.py` draws two slim translucent bars that the animator composites onto every output frame: a **header** under the window chrome (title · current section) and a **footer** at the very bottom edge carrying `author` · `source`, a URL, an optional running clock, and a small persistent watermark. Both are driven by the same `MetadataConfig` that feeds the container tags and the steganographic payload, so attribution is set once and stamped in three places — the full three-carrier model is developed in sec. 11. The footer (with its clock) is on by default; the header is opt-in. Like every other overlay the bars scale with the frame height and no-op silently when their fields are empty.

6.8 Export: Chapters, Posters, and GIFs

A set of deliverables widens the output beyond the MP4, HTML player, and captions. `export/chapters.py` derives YouTube-style chapter markers from the demo's scene boundaries (`to_youtube_chapters`) and an `ffmetadata` block (`to_ffmetadata`) that `embed_chapters` muxes into the MP4 with `ffmpeg` — verified by reading the embedded chapter entries back with `ffprobe` (sec. 8) — so a long demo arrives already chaptered. Crucially, the chapter timestamps are taken from the *measured* timeline (sec. 4), not from the pre-render word-count estimate: each chapter begins at the cumulative measured duration of the scenes before it, so the chapter markers land exactly on the scene transitions the viewer sees. The two produced videos carry one chapter per scene — 14 for the package showcase and 13 for the paper demo (sec. 8) — aligned to the same clock that drives the waveform and the action timestamps. `export/poster.py` renders a standalone thumbnail/poster (`render_poster`) and a short looping preview GIF (`demo_to_gif`) sampled from the rendered frames. The CLI surfaces these as `democreate thumbnail` and `democreate gif`. As with the rest of the pipeline, the chapter, poster, and GIF paths degrade gracefully: the deterministic core still yields the player, transcript, thumbnail, and GIF when `ffmpeg` is absent, and only the embedded-chapter mux requires the binary.

6.9 Audio: Pacing, Normalization, and Fades

The `AudioConfig` controls the voiceover's pacing and loudness. `lead_silence_ms`, `gap_ms`, and `trail_silence_ms` set the silences that `concat_with_gaps` inserts around and between chunks; crucially, the animator's timeline reserves the *same* silences, keeping audio and video in lock-step (sec. 4). `normalize` applies EBU R128 loudness normalization (European Broadcasting Union 2020) via `ffmpeg`'s `loudnorm` filter (FFmpeg Developers 2024a) — defaulting to an integrated −16 LUFS target with a −1.5 dBTP true-peak ceiling — so narration assembled from clips of varying loudness plays back at a consistent, broadcast-sane level; the worked example in sec. 7 verifies at a measured −15.5 dB mean volume, on target. `fade_ms` adds a guarded fade-in and fade-out. As with every other backend dependency, the loudness and fade steps are guarded: when `ffmpeg` is absent the pipeline still produces a valid, concatenated voiceover from the pure-stdlib path, simply without the normalization pass — fidelity degrades, capability does not.

7 Research-Paper Demos

DemoCreate's second domain is the research paper. A scientific result typically ships as a PDF *and* a codebase, and a good overview of it walks the reader through the title and authors, the abstract, the key figures, and the architecture of the implementation that produced them. The `paper/` subsystem treats that overview as exactly the same compilation problem as a software tour: a PDF (plus its figures and codebase) is read into a structured summary, and `build_paper_demo` composes that summary into an ordinary `Demo`

value — after which the *identical* narration, synchronization, animation, and export pipeline (sec. 3) renders it. Nothing in the paper path is a special case downstream; it is content authored into the same spine. fig. 8 shows the whole path end to end.

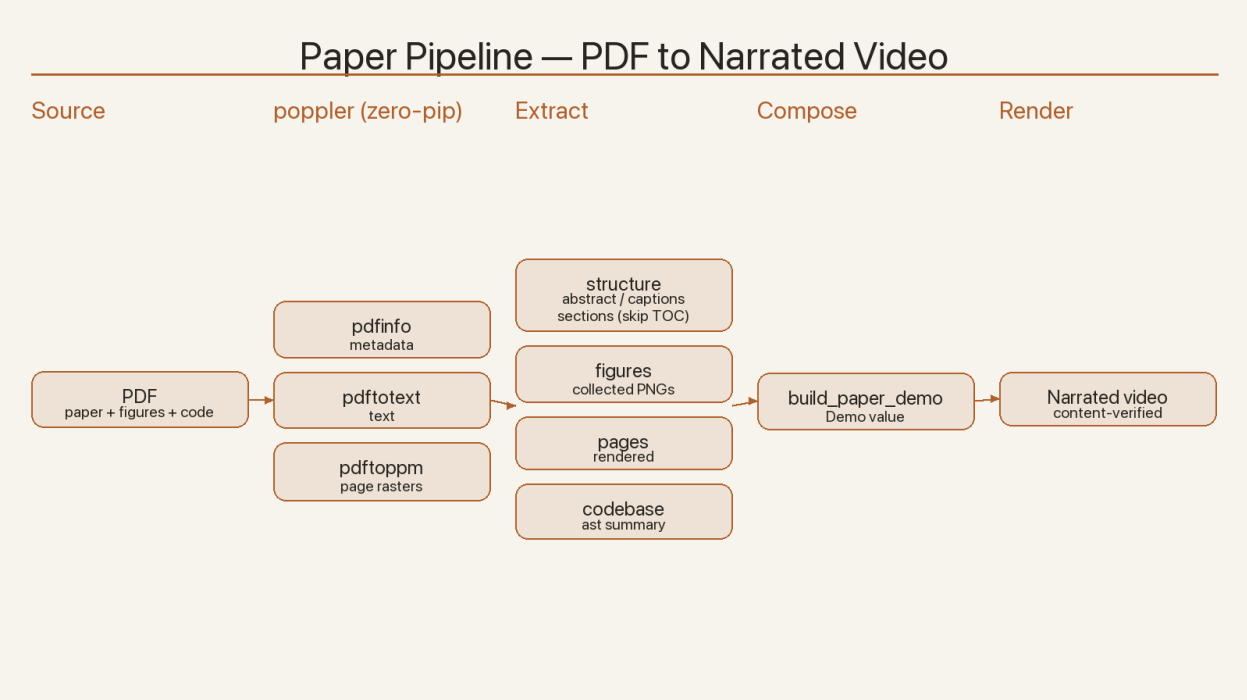

Figure 8: The paper pipeline, drawn under the `paper` theme by the same diagram renderer the package ships (sec. 5): a PDF flows through the three poppler binaries (`pdfinfo`, `pdftotext`, `pdftoppm`), which feed the structure extractor (abstract, figure captions, and sections — explicitly skipping the table of contents), the collected figure PNGs, the rendered pages, and the `ast` codebase summary; `build_paper_demo` composes these into a `Demo` value that the standard pipeline renders into a content-verified narrated video.

7.1 Reading a Paper with Zero Pip Dependencies

The PDF is read through the poppler command-line utilities (The Poppler Developers 2024), not a Python PDF library, so paper support adds *no* pip dependency to a workstation that already has poppler installed. `paper/pdf.py` wraps three binaries: `pdfinfo` for metadata, `pdftotext` for text extraction, and `pdftoppm` for rasterizing pages to PNG images. Each call first checks the binary is on `PATH` and raises `BackendUnavailableError` (naming `poppler` and the `pdf` extra) otherwise, so a missing tool yields an actionable message rather than an opaque error. Every call is deterministic, read-only, and operates on the real file.

`paper/extract.py` turns those calls into a render-ready `PaperSummary`: title and authors from `pdfinfo` metadata, the page count, and a sorted list of figure images collected from a sibling directory. Figures are *collected*, not extracted from the PDF: the subsystem takes a directory of already-exported figure PNGs (the reproducible analysis outputs the project already has), which keeps figure quality high and the implementation simple.

7.2 Real Structure: Abstract, Captions, and Sections — Skipping the TOC

The depth of a paper demo comes from `paper/structure.py`, whose three pure extractors read the paper's *real* prose rather than generic boilerplate, and whose recurring adversary is the table of contents (TOC) that long manuscripts front-load. The extractors operate on already-extracted text, so they are fully testable without poppler; only `summarize_structure` touches the guarded poppler path.

`extract_abstract` finds the *real* abstract, not the title block. A naive `/abstract/i` match fails on a long paper because the TOC contains an "Abstract" entry before the body does. The extractor instead iterates over *every* standalone `Abstract` marker and returns the first following block that is substantial (≥200 characters) and is *not* itself a TOC — TOC slices are detected by their dotted leaders (`....`) and trailing page numbers and rejected outright. If no marker yields a clean block it falls back to the first substantial prose paragraph past the front matter, never the repeated title. `extract_figure_captions` recognizes real caption lines (`Figure 3: …`, `Fig. 3 …`), captures the *multi-line* caption body — joining the continuation lines that a PDF text dump wraps a long caption across — bounds it to a sentence under a character budget, deduplicates by figure number, and discards cross-reference and TOC fragments. All three extractors first pass their input through an **NFKC glyph fold** that normalizes the ligatures, smart quotes, soft hyphens, and full-width punctuation that `pdftotext` emits, so a caption or heading is matched on its canonical characters rather than failing on a typographic variant. `extract_sections` recognizes numbered headings (`2.1 Setup`), `Part N - Title` headings, and a fixed set of named sections (Abstract, Introduction, Methods, …), again skipping any line that carries dotted leaders or a trailing page number. On the *Policy Entanglement in Active Inference* paper these extractors yield the correct ~**1200-character abstract** and the paper's real **6-part section structure** (Introduction, Theory, Formal Verification, Empirical Grounding, Connections to Existing Frameworks, Discussion and Outlook) — real structure read straight from the document, with the TOC reliably rejected throughout, plus a deduplicated set of real figure captions that drive the per-figure narration below.

7.3 Composing the Paper Demo

`paper/script.py`'s `build_paper_demo` assembles the summary into a sequence of scenes: a **title** card (paper title and author line), an optional **title-page** background from the rendered first PDF page, the **abstract** paced into narration-sized chunks at sentence boundaries by `chunk_sentences`, a **structure** scene that names how the paper is organised (the real section list from `extract_sections`, e.g. "The paper is organised into *N* parts: …"), each selected **figure** as a full-frame background with a "Figure *n*" section pill — narrated with its *real* caption from `extract_figure_captions` when one is available, falling back to a generic line otherwise — a few additional **PDF pages** rendered directly from the document, a **codebase architecture** slide (a diagram of the paper's implementation, grouped by top-level directory), and a closing **outro** that states the reproducibility thesis. The structure scene and the real-caption figure narration are the depth that makes a paper demo specific: it describes *this* paper's actual sections and figures rather than narrating them generically. fig. 9 shows the central composition: a real published figure fit *whole* into the frame under the `paper` theme, with the section pill in the chrome and the narration caption in its own band below — exactly the no-crop slide `build_paper_demo` emits for each figure.

The narration is template-driven and deterministic — no LLM is required — so the same paper always yields the same demo; `narration_overrides` allow customization. Because the output is an ordinary `Demo`, the paper theme's warm slides, the moving waveform, scene crossfades, and the no-crop fit-whole layout for every figure and page (sec. 6) all apply for free, and the result is content-verified by the same `verify_video` check as any other render (sec. 8). The CLI exposes the whole path in one command:

```
democreate paper paper.pdf --repo ./code --figures ./figures --theme paper
```

7.4 Worked Example: *Policy Entanglement in Active Inference*

The paper path was exercised end to end on a real, substantial artifact: the *Policy Entanglement in Active Inference* paper — a 170-page PDF — and its 145-module codebase. DemoCreate read the PDF's metadata through poppler and ran the structure extractor over its text, recovering the real ~**1200-character abstract** (skipping the front-matter TOC) and the paper's real **6-part section structure** — which drove a structure scene and per-figure narration from the real captions. It rendered selected pages to images, collected the figure PNGs from the project's reproducible analysis outputs, walked the codebase with the stdlib `ast` summarizer (sec. 5) and rendered its architecture as a diagram, and composed the result into a 12-scene `Demo` (title, front page, abstract, section structure, six captioned figures, the codebase architecture, and an outro). The pipeline then synthesized narration, measured its timing through the TTS→STT round-trip,

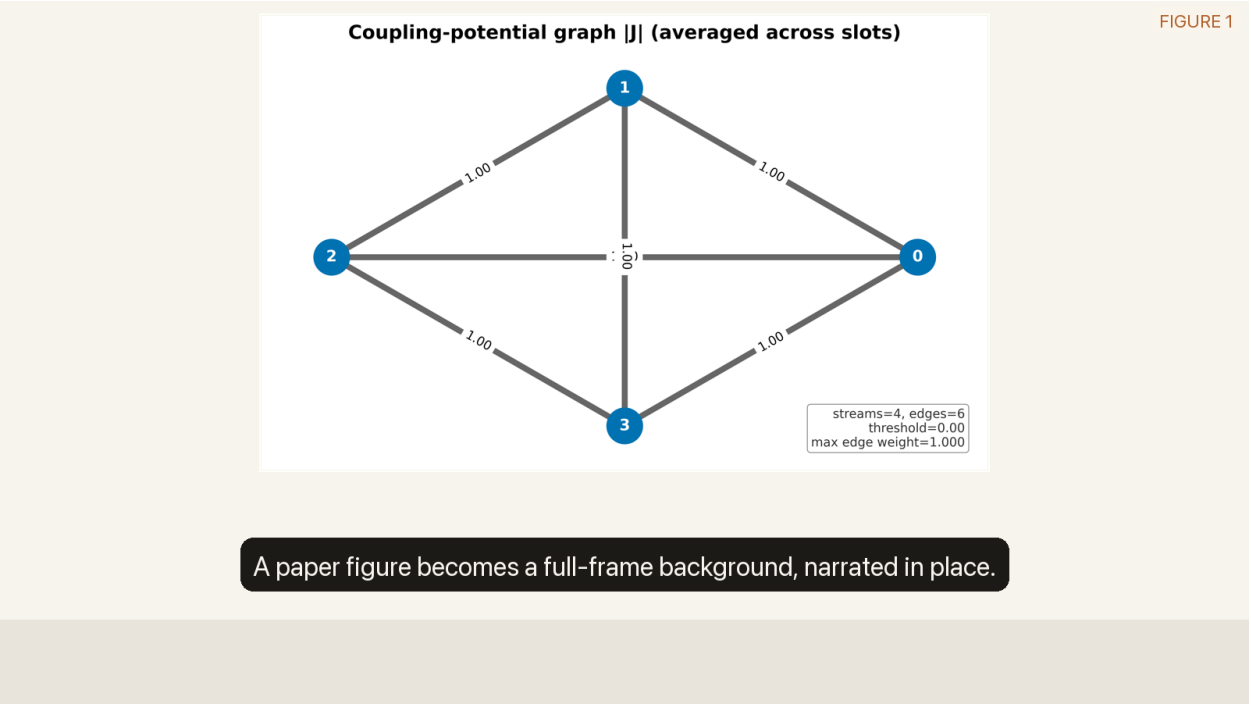

Figure 9: A paper-figure slide as composed by `build_paper_demo` and rendered by `SyntheticRenderer` under the `paper` theme: a real published figure (a coupling-potential graph from the *Policy Entanglement* paper) is fit *whole* into the content area as a `background_image` (contain, never cropped), with a "Figure 1" section pill in the chrome and the narration caption in a band below the image. The no-crop layout (sec. 6) keeps the entire figure — axes, labels, and all — visible.

animated the frames with the typing reveal, the animated cursor, the moving waveform, and scene transitions (sec. 6), fit every figure and page *whole* into the frame under the no-crop layout, normalized the voiceover to EBU R128, and encoded the whole thing with `ffmpeg`.

The output was a single **1920×1080**, **188.0-second**, **H.264** MP4 with a real voiceover normalized to a measured **−15.5 dB** mean volume — on the −16 LUFS target (sec. 6) — and the content verifier confirmed it carries genuine, non-silent, non-black video and audio streams (sec. 8). This is the central claim of the subsystem made concrete: a real paper — its true abstract, six captioned figures, and 6-part section structure — and its codebase compiled, with no neural model and no proprietary PDF library, into a narrated, verified, reproducible overview video.

8 Evaluation: Measured Performance, Testability, Verification, and Determinism

DemoCreate makes no empirical claims about human preference or production quality; the appropriate evaluation for a package of this kind is of its *engineering* properties. We assess five: that the build and render stages are *fast enough* to use interactively, measured rather than asserted; that the core is testable without heavy backends; that it reaches a high coverage gate against real artifacts rather than mocks; that rendered video is *content-verified* rather than merely produced; and that its output is reproducible by construction.

8.1 Measured Performance

The figures below are *measured*, recorded in the repository's `data/benchmarks.json` and reproducible with the package's own timing harness — not back-of-the-envelope estimates. tbl. 3 collects them, and fig. 10 charts the two latency numbers.

Table 3: Real measured benchmarks from `data/benchmarks.json`. The build median is over five runs of the default starter-demo pipeline; the render throughput is compute-time per second of finished video at the animation frame rate; the sync row confirms the audio-anchored timestamps are complete and monotonic.

Metric	Measured value	What it covers
Build pipeline (median of 5 runs)	**25.8 ms**	`build_demo` end to end on the starter demo: schema construction, validation, and manifest assembly.
Animated render	**256.7 ms** of compute per second of output (at 15 fps)	The animator's per-frame compositing — typed re-renders, waveform, progress, cursor, transitions — over a 7.72 s clip (1981.9 ms total for 7.72 s of output).
Synchronization timestamps	**3 of 3 actions** timestamped, **monotonic**	Every action's trigger word resolved to a strictly non-decreasing spoken time across the measured timeline.

The build median of 25.8 ms means a `Demo` value is constructed, self-validated, and laid out into a render manifest faster than a single 60 fps display frame — the declarative spine is effectively free, so iteration is interactive. Rendering is the cost center, as expected of a frame-compositing pipeline, but at 256.7 ms of compute per second of finished video at 15 fps it is comfortably faster than real time per frame and embarrassingly parallel across frames. The synchronization row is the qualitative guarantee made quantitative: all three actions in the benchmark demo resolved to timestamps, and those timestamps are monotonic — the audio-anchored ordering (sec. 4) holds on real data, not just in the type system.

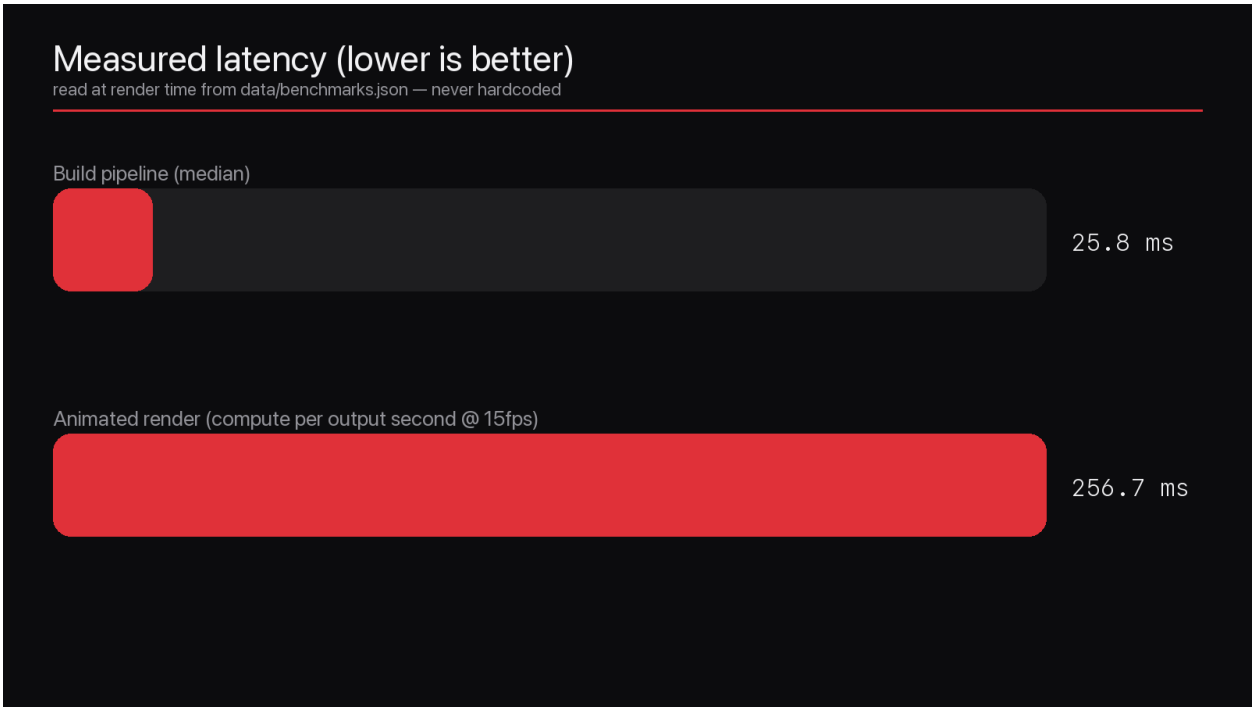

Figure 10: Measured latency from `data/benchmarks.json`, drawn directly by the manuscript's figure script with no plotting dependency: the build pipeline's 25.8 ms median dwarfed by the 256.7 ms of render compute per second of output at 15 fps. Lower is better; the build stage is interactive, and render dominates the wall-clock cost as expected of a frame compositor.

8.2 A Pure, Testable Core

The central design decision — every heavy backend behind an interface with a pure-Python deterministic default (sec. 3) — is what makes the package testable at all. Because the silent TTS backend, the heuristic transcriber, the synthetic renderer, and the manifest compositor are all stdlib-or-Pillow implementations that produce *real* files, the entire pipeline can run, end to end, in an environment with none of `kokoro`, `whisper`, `mss`, `playwright`, `manim`, or even `ffmpeg` installed. The test suite therefore exercises genuine behavior: it writes real WAV files and reads their real durations back, it draws real PNG frames and asserts on their pixels, it serializes and re-parses real JSON and YAML, and it asserts on the bytes and structures that result. The thin adapters to heavy backends are the *only* code excluded from measurement, and they are excluded precisely because they cannot execute without proprietary or binary dependencies — they are guarded by constructors that raise `BackendUnavailableError` and are annotated `pragma: no cover` only along the unreachable synthesis path.

The suite comprises **625 passing tests** (with 3 skipped where an optional binary is absent) across more than three dozen test modules spanning the package's 51 source modules in 7 subsystems: schema round-tripping and validation, config and theme serialization (including the aspect-ratio presets and the resolution tiers), TTS synthesis and duration accuracy, the TTS→STT round-trip and trigger-word anchoring, gap-aware timeline construction, the typing-reveal and animated-cursor compositing, the no-crop background fit and code-autosize-and-wrap paths, the on-screen overlay bars and clock formatting, audio concatenation and the guarded normalization path, waveform envelope computation, the diagram and synthetic renderers, caption formatting, codebase summarization, the poppler-backed paper subsystem and its real abstract/-caption/section extraction, the chapter, poster, and GIF export paths, the container-metadata tag builder, the LSB steganography round-trip and signed-provenance verification, the `ffprobe`-parsing verifier, export rendering, the CLI commands, and the end-to-end pipeline. Tests follow a real-filesystem discipline in the spirit of xUnit test patterns (Meszaros 2007) and run under pytest (pytest-dev team 2024): they create

temporary workspaces, run real stages against them, and inspect the produced files, rather than asserting against stubbed return values.

8.3 A Coverage Gate Against Real Artifacts

Coverage is enforced as a gate, not reported as a vanity metric. The `pyproject.toml` configuration sets a $\geq$**90%** `fail_under` threshold on coverage of `src/democreate`, so a build that drops the pure core below ninety percent fails; the suite clears it comfortably at roughly **95%**. The exclusion list is narrow and principled — `pragma: no cover` (the guarded backend bodies), `raise NotImplementedError` (abstract method surfaces), `if __name__ == "__main__"` (console entry points), and `if TYPE_CHECKING` (import-only typing blocks). Crucially, the gate is measured against the deterministic default path, which is the path real users exercise when they install only the core dependencies. It therefore certifies the *actually-reachable* code, not an artificially inflated number padded by mock-driven tests of unreachable branches.

8.4 Content-Asserting Verification

Producing a video file is not the same as producing a *good* video file. An all-black clip, a single-frame stub, or a video carrying a silent audio track all satisfy a naive "does `ffprobe` report a 1920×1080 stream?" check. `export/verify.py` refuses that bar. `verify_video` asserts, against the real file, that there is a video stream of the expected dimensions and at least the expected duration; a second, *audio* stream whose duration covers the video; that the audio is **not digital silence** (mean volume above a −60 dB floor, measured with `ffmpeg`'s `volumedetect`); and that at least one sampled mid-stream frame is **not uniformly black** (grayscale pixel variance above a floor). The JSON-parsing core, `parse_ffprobe`, is pure and unit-tested with canned `ffprobe` output; only the probes that shell out to the binaries are guarded and excluded from coverage. The CLI surfaces this as `democreate verify`, and both `render` and `paper` run it automatically, failing the command if the assertions do not hold.

8.5 Tamper-Evident Provenance and 4K Geometry

Two distribution properties are likewise verified rather than asserted. The first is the **steganographic round-trip and its tamper-evidence** (sec. 11). A test embeds a signed provenance record into a lossless poster PNG, extracts it back, and confirms the recovered JSON is byte-for-byte the record written; it then calls `verify_provenance` against the *original* demo and gets `True`, and against a demo whose content has been edited gets `False`. The verification is robust to render-state churn precisely because `_content_digest` excludes `audio_path/start_ms/timestamp_ms/duration_ms`, so a re-rendered demo with fresh timestamps still verifies `True` — the digest reacts only to *authored* change. This makes "the PNG matches this demo" a checkable property, and "this demo was edited" a detectable one. The second is **resolution geometry**: selecting the `1440p` tier via `set_resolution` produces a stream measured at a true 2560×1440, and `2160p/4k` at 3840×2160, confirming that the height-relative renderer scales to distribution resolutions rather than upscaling — a higher tier is genuinely more pixels.

8.6 Two Real Rendered Videos

The end-to-end claims are backed by two real renders, not just unit tests, **both re-rendered in the v0.6.2 noir aesthetic and re-verified**. A **software** render exercised the full `render` path — synthetic frames with the character-by-character typing reveal and animated cursor (sec. 6), a real OS voiceover, the animator's moving waveform and top-edge progress line, and an `ffmpeg` encode — to a content-verified MP4. A **research-paper** render (sec. 7) compiled the *Policy Entanglement in Active Inference* paper — a 170-page PDF from which the structure extractor read a real abstract, figure captions, and the paper's **6-part section structure** (sec. 7) — into a **1920×1080, 188.0-second H.264** video that the verifier confirmed carries real, non-silent, non-black streams. Both passed `verify_video` end to end, demonstrating that the deterministic core and its light backends produce genuinely playable, verified deliverables — not file-existence stubs.

8.7 Produced Demonstrations

The two videos exist as concrete artifacts, both re-rendered in the v0.6.2 noir aesthetic, and stills lifted directly from the encoded files appear in fig. 11. The **canonical package demo** is DemoCreate's *showcase* (`output/video/demo.mp4`), a 1920×1080 H.264 clip of **128.4 s** compiled from one declarative file (`examples/democreate_showcase.json`) that exercises *every* renderable surface across **14 scenes**: a hero title card, the one-glance graphical abstract, the declarative-spine, backends, and synchronization code typing in character by character, the **five-theme strip** (noir is the default), a real published research-paper figure fit *whole* into the frame, the architecture diagram, a terminal render-and-verify, an outro, and **bullet slides** (`FrameState.bullets`, e.g. *"A demo is a value, not a recording"*) and a **stat-card slide** (`FrameState.stats`, *625 tests · 7 subsystems · 5 themes · 4K · 0 pip*). It carries 14 embedded chapters (one per scene) and the container tags `title="DemoCreate - The Showcase"` / `artist="Daniel Ari Friedman"`. The **research-paper demo** (`output/paper_demo/video/demo.mp4`) is a 1920×1080, **188.0 s** render of the *Policy Entanglement in Active Inference* paper — also in noir (its warm paper theme was replaced) — with its real abstract paced across chunks, the 6-part section structure, six figures with their real captions, and a codebase-architecture diagram, across 12 chaptered scenes. Both were **content-verified** by `verify_video` (real, non-silent, non-black streams; 48 kHz audio) and ship with the full distribution surface from sec. 11: per-video YouTube **chapters**, MP4 **container tags** (`title`/`artist`), and a **signed steganographic provenance poster**. (An earlier intro demo still renders but is superseded by the showcase as the canonical package demo.) The `output/` tree is gitignored because the videos are regeneratable from their declarative sources by the exact `render` and `paper` commands above; the stills in fig. 11 are checked in as durable evidence of the produced renders.

Figure 11: Stills lifted directly from the two encoded videos (1920×1080 each), both re-rendered in the v0.6.2 noir aesthetic — near-black surfaces, white text, a single red accent, and a top-edge progress line. Top row — the package showcase (`output/video/demo.mp4`, 128.4 s): the "by the numbers" stat-card slide and a code scene typing in character by character (monochrome with red keywords). Bottom row — a research-paper demo: an abstract scene and a real published figure fit *whole* into the frame under the no-crop layout. Each is real evidence of a content-verified render, not a mock-up.

8.8 Reproducibility by Construction

The strongest determinism guarantee follows from the spine itself. Rendering is a pure function of the `Demo` value, and every default backend is deterministic: the silent TTS backend writes the same silence for the same text, the heuristic transcriber distributes words by a fixed length-weighting over a measured duration, the timeline builder lays chunks back-to-back by a fixed rule, and the synthetic renderer draws the same frame for the same `FrameState` and `Theme`. There is no random seed to fix, no model weights to pin, and no wall-clock dependence in the default path. Consequently the equality `Demo.from_dict(d.to_dict()) == d` holds by construction, and a given `Demo` re-renders to byte-stable manifests and frames on any machine with the core dependencies — the very property the manuscript figures rely on, since each was produced by calling the same public renderers. This is reproducibility *by construction* rather than by convention, which is exactly the gap that captured, non-re-executable artifacts leave open (Pimentel et al. 2019). When the heavy backends are installed, determinism is necessarily bounded by the determinism of the underlying neural models, but the orchestration, the timeline, and the declarative source remain reproducible, and the demo can always be re-rendered in the deterministic default mode for a stable baseline.

9 Reproducibility and Use

This section records the concrete steps to install DemoCreate, run its test gate, and produce a demo — of either software or a research paper — so that the claims of sec. 8 are independently checkable.

9.1 Environment and Installation

The package targets Python 3.10 and later and is managed with uv (Astral 2024) for reproducible, lockfile-driven environments. The core dependencies are deliberately light — `pyyaml`, `typer`, `rich`, `jinja2`, and `pillow` — so an editable install pulls no heavy binaries:

```
uv venv
uv pip install -e ".[dev]"
```

Optional heavy backends are installed only as needed, each named for the subsystem it upgrades:

```
uv pip install -e ".[tts]"        # Kokoro / Chatterbox neural TTS
uv pip install -e ".[whisper]"    # Whisper word-level transcription
uv pip install -e ".[capture]"    # mss real-pixel screen capture
uv pip install -e ".[browser]"    # Playwright website driving
uv pip install -e ".[animation]"  # Manim animation
uv pip install -e ".[video]"      # MoviePy / ffmpeg-python helpers
uv pip install -e ".[all]"        # everything
```

Two high-fidelity capabilities need no pip extra at all, only system binaries that are commonly present: real spoken narration via the operating system's `say`/`espeak` (the `SystemTTSBackend`, sec. 3), and research-paper ingestion via the poppler utilities (The Poppler Developers 2024) (sec. 7). The final video encode and the EBU R128 loudness pass (sec. 6) use `ffmpeg`. The `democreate backends` command reports, at runtime, which extras and binaries are present and which capabilities are running on their deterministic default — confirming that every capability has a working default backend.

9.2 The Test Gate

The test gate is the primary reproducibility check. It runs the full 625-test suite (3 skipped where an optional binary is absent) against real artifacts and enforces the $\geq$90% coverage threshold configured in `pyproject.toml`, which the suite clears at roughly 95%:

```
uv run pytest --cov=src/democreate --cov-report=term-missing
```

Because the suite is pure-Python and uses no mocks, it requires only the core and `dev` dependencies; it does not need any heavy backend installed. Tests that would exercise an optional heavy backend skip cleanly

when the extra is absent. Static checks — `ruff` for linting and import ordering, `mypy` for typing against the shipped `py.typed` marker — complete the gate.

9.3 Producing a Software Demo

The CLI is a thin orchestration layer over the library; every command resolves to a few calls into the pipeline and subsystems. Its commands are `init`, `inspect`, `build`, `tour`, `captions`, `render`, `verify`, `paper`, `backends`, and `version`. The quickest path from nothing to an inspectable artifact is:

```
democreate init demo.json          # write an editable starter demo
democreate inspect demo.json       # validate and summarize it
democreate build demo.json -o out  # run the full pipeline
```

`build` runs validation, TTS, TTS→STT synchronization, timeline resolution, compositing, caption emission, and export, writing audio, frames, a render manifest, SRT/VTT captions, a Markdown transcript, the serialized demo JSON, and a self-contained interactive HTML player under `out/`. To produce and *content-verify* an HD MP4 with a real voiceover and a chosen theme:

```
democreate render demo.json -o out --tts system --voice Samantha --theme midnight
democreate verify out/<video>.mp4 --width 1920 --height 1080
```

A complete codebase tour can be generated and rendered directly from a repository:

```
democreate tour /path/to/repo -o out --title "My Project Tour"
```

This walks the repository's Python sources with the stdlib `ast`-based summarizer, generates a declarative `Demo` via `generate_codebase_demo`, writes it to `out/demos/tour.json`, and renders it.

9.4 Producing a Research-Paper Demo

The paper path (sec. 7) is a single command from a PDF (and optionally its figures and codebase) to a verified video:

```
democreate paper paper.pdf --repo ./code --figures ./figures --theme paper
```

This reads the PDF through poppler, extracts the title, authors, the real abstract, figure captions, and sections, collects the figure images, walks the codebase for an architecture diagram, composes a `Demo`, renders it, and runs `verify_video`. The *Policy Entanglement* worked example of sec. 8 is reproduced by exactly this invocation.

Subtitles for any demo can be emitted to standard output in SRT, VTT, or ASS form via `democreate captions demo.json --format vtt`. Worked examples accompany the package under `examples/`, and because a demo is a plain JSON or YAML value, every example — and every figure in this manuscript — is itself a reproducible input that re-renders identically (sec. 8).

10 Scope and Related Work

DemoCreate occupies a specific niche — declarative, deterministic, backend-pluggable generation of narrated demos of *both* software and research papers — and it is honest about what neighboring systems do better. This section compares it against its prior art along the dimensions that matter: the content model (captured pixels, as in capture-first screen recorders such as Recordly (Recordly 2024), versus a declarative description), the synchronization strategy, the dependency footprint, the breadth of demo kinds supported, and — the distinctive axis — whether the system can demo a *research paper* reproducibly.

10.1 Event-Sourced Virtual IDEs

CodeVideo (CodeVideo 2024) is the closest conceptual relative and an explicit influence. Its core insight — model an IDE as a deterministic state machine driven by an ordered stream of typed actions ("IActions"),

so a coding tutorial is a replayable log rather than a recording — is precisely the event-sourcing discipline (Fowler 2005) that DemoCreate adopts for its `Action` stream. CodeVideo is a mature TypeScript ecosystem with a polished web studio and a rich virtual-editor model; DemoCreate is a Python library and does not match its front-end. What DemoCreate adds on top of the shared action-stream idea is the *narration-first synchronization layer* and the *research-paper domain*: CodeVideo's model is fundamentally visual and code-only, whereas DemoCreate subordinates actions to spoken words, pins them to *measured* audio timestamps via a TTS→STT round-trip (sec. 4), and extends the same spine to PDF-backed paper demos (sec. 7).

VSpeak (VSpeak 2024) contributes the other half of DemoCreate's spine: the chunk-and-trigger narration model, in which narration is the organizing unit and on-screen events are anchored to spoken trigger words. DemoCreate's `Chunk` and the per-action `trigger_word` are a direct descendant. Where DemoCreate extends VSpeak is in resolving the trigger word to an *absolute* timestamp by transcribing real audio rather than relying on a prearranged timing — VSpeak names the anchor; DemoCreate measures where the anchor lands.

10.2 Terminal Recorders

asciinema (Kulik and asciinema contributors 2024) and **termtosvg** (Bedos 2020) occupy a narrower but durable corner: deterministic terminal capture. asciinema records a session as a timed event stream in the asciicast v2 format — a model so sound that DemoCreate reuses it directly in `capture/terminal.py` to represent `terminal` scenes without launching a shell. termtosvg renders such recordings to standalone animated SVG. Both are excellent at exactly one demo kind (the terminal) and make no attempt at narration, multi-surface scenes, or audio synchronization. DemoCreate is broader — editor, website, terminal, slide, and paper-figure scenes in one artifact — and narrated, but it does not aim to replace asciinema for pure terminal sharing; it embeds asciinema's model as one capture strategy among several.

10.3 Code-Animation Engines

code-video-generator (Wood and code-video-generator contributors 2023) scripts Manim (The Manim Community Developers 2024) to animate code walkthroughs. It is focused and effective for animated code, but it is Manim-centric — animation is mandatory rather than optional — and it addresses neither narration synchronization nor terminal, website, or paper scenes. **Manim** itself is a general-purpose programmatic animation engine, superb for mathematical exposition and bespoke motion, but it is a heavy dependency and a low-level drawing API, not a demo compiler: there is no notion of a declarative demo, of narration chunks, or of audio-anchored timing. DemoCreate treats Manim as one *optional* animation backend behind the `animation` extra, with a pure-Pillow synthetic renderer and timed-frame animator as the always-available default — so motion (waveform, transitions, Ken Burns) is present in the core build with no Manim installed.

10.4 Agentic and Paper-to-Video Generators

Code2Video (Chen et al. 2025) and **Paper2Video** (Zhu et al. 2025) are recent agentic, LLM-driven systems, and Paper2Video is the most direct comparison for DemoCreate's paper domain. Code2Video uses a Planner–Coder–Critic agent loop to generate educational videos by emitting and executing Manim code; Paper2Video turns a scientific paper into a narrated presentation video with synchronized slides, speech, and a talking presenter. Both are powerful and ambitious, and both are fundamentally *generative and non-deterministic*: their quality depends on large models, their output varies across runs, and they carry heavy dependencies as a baseline. DemoCreate makes the opposite bet for the same paper-to-video goal. Its `paper/` subsystem reads the PDF with poppler (The Poppler Developers 2024), composes a deterministic `Demo` with template narration, and compiles it with a pure default pipeline — *no LLM, no neural model required*, and the same paper always yields the same verified video (sec. 7, sec. 8). Paper2Video aims for a richer, presenter-style artifact; DemoCreate aims for a reproducible, dependency-light overview that re-renders identically and is content-verified by construction. The systems are complementary rather than competing: an agentic planner in the Code2Video or Paper2Video mold is a natural *producer* of the declarative `Demo` that DemoCreate then renders reproducibly.

10.5 A Capability Comparison

tbl. 4 places DemoCreate beside its closest neighbors along the dimensions that distinguish the niche. The cells are honest: a checkmark means the system makes the property a core, default guarantee; a dash means it does not (often by design, as for asciinema's deliberate terminal-only focus). The pattern is the point — no single peer holds the whole row, and several columns (the deterministic default, audio-anchored sync, and reproducible research-paper demos together) are held only by DemoCreate.

Table 4: DemoCreate versus prior art across the six dimensions that define its niche. "partial" on VSpeak's sync row reflects that VSpeak *names* a trigger word but does not measure where it lands; "partial" on Paper2Video's sync row reflects that it produces speech aligned to slides generatively but does not anchor actions to *measured* word onsets via a TTS→STT round-trip; "manual" on Manim's typing row reflects that typing must be hand-animated rather than emerging from the model.

Capability	DemoCreate	CodeVideo	VSpeak	asciinema	Paper2Video	code-video-gen.	Manim
Declarative spine	✓	✓	✓	✓ (asciicast)	–	–	–
Deterministic by default	✓	✓	✓	✓	– (LLM)	✓	✓
Audio-anchored sync (TTS→STT)	✓	–	partial (trigger only)	–	partial (synth, not STT)	–	–
Typing animation	✓	✓	–	✓ (replay)	–	✓	manual
Research-paper demos	✓	–	–	–	✓	–	–
Zero-pip defaults	✓	n/a (TS)	–	✓	–	– (Manim)	–

10.6 Positioning

DemoCreate's contribution is not any single capability but their conjunction in one library. Four properties hold together: a declarative, self-validating, losslessly-serializable spine that fuses CodeVideo's event sourcing with VSpeak's chunk-and-trigger narration; a backend-abstraction discipline in which *every* heavy capability has a working pure-Python default, so the package produces a real, content-verified demo with five light dependencies and upgrades in fidelity rather than capability; audio-anchored synchronization that *measures* action timing from a TTS→STT round-trip rather than estimating it; and — the property with the least prior art outside the heavy agentic systems — *deterministic, dependency-light demos of research papers*, not just software. No single peer holds all four (tbl. 4). The honest summary is that DemoCreate is narrower than the agentic generators in raw output ambition and less specialized than asciinema for pure terminals — but it is the only system of the set whose demos are simultaneously declarative, deterministic, narrated, multi-surface, paper-capable, content-verified, and runnable with no heavy backend at all.

11 Provenance and Distribution

A generated demo is meant to be *distributed* — posted, embedded, forwarded, re-uploaded — and once it leaves the machine that rendered it, two questions follow it everywhere: *who made this, and is it the artifact they made?* DemoCreate answers both by stamping provenance onto the render through three independent

carriers, all driven from a single `MetadataConfig` (sec. 3), and by raising the output's resolution and quality to genuine distribution grade. Because the three carriers degrade independently, provenance survives the transforms a video undergoes in the wild even when any one carrier is stripped.

11.1 The Three-Carrier Model

`MetadataConfig` is a plain dataclass folded into `RenderConfig` alongside `Theme`, `AudioConfig`, and `VideoConfig`. Its fields — `author`, `title`, `date`, `source`, `url`, `watermark` — are *content*, not styling, and the same values feed three different carriers, so a creator sets their attribution once and it propagates everywhere. fig. 12 lays the model out: one config, three carriers, each with its own survivability property.

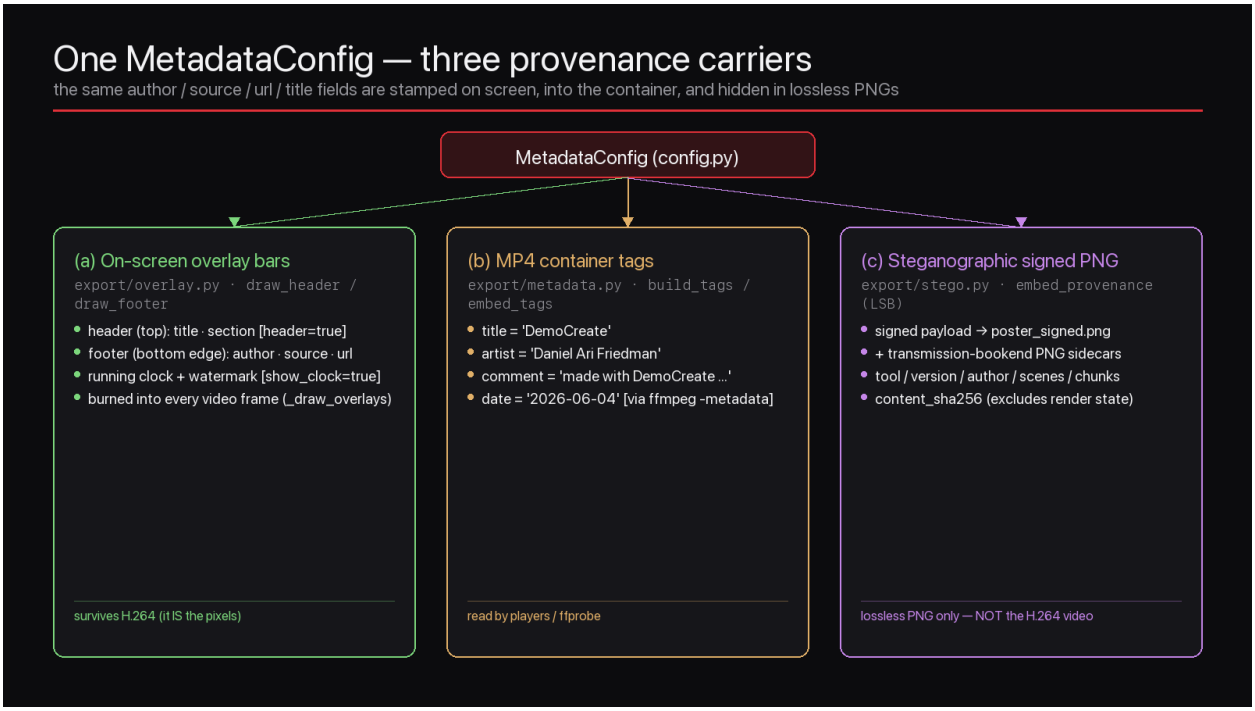

Figure 12: One `MetadataConfig` drives three independent provenance carriers. **(a)** On-screen overlay bars (`export/overlay.py`) burn a header (title · section) and a footer (author · source · url · running clock · watermark) into every video frame — provenance that *is* the pixels and therefore survives any re-encode. **(b)** MP4 container tags (`export/metadata.py`) write `title`/`artist`/`comment`/`date` into the file via `ffmpeg -metadata`, read back by players and `ffprobe`. **(c)** A signed steganographic payload (`export/stego.py`) hides a content-hashed provenance record in lossless poster and transmission-bookend PNG sidecars — recoverable from the PNGs, but **not** from the H.264 video pixels. The same author/source/url/title fields populate all three.

On-screen overlay bars. `export/overlay.py` draws two translucent broadcast-style bars and the animator composites them onto every output frame via `_draw_overlays`. `draw_header` pins a slim ribbon just under the window chrome carrying the title (left) and the current section (right); `draw_footer` pins a strap at the very bottom edge carrying `author · source`, the URL, an optional running clock formatted `M:SS / M:SS` by `format_clock`, and a small persistent watermark at the far right. The footer is on by default (`footer=True`, `show_clock=True`); the header is opt-in. Because the bars are drawn proportionally to the frame height they read correctly at any resolution, and because they are *part of the rendered pixels* they are the one carrier that survives an H.264 encode intact — the trade-off being that they are visible by design.

Container tags. `export/metadata.py` turns the demo plus its `MetadataConfig` into standard MP4 metadata. `build_tags` is a pure function producing the canonical `title` (the metadata title overriding the demo title), `artist` (the author), `date`, and a `comment`/`description` credit line ("made with DemoCreate `version`

· source: … · url"), dropping any empty value. The pipeline's `_embed_provenance` step calls `embed_tags`, which stream-copies the video through `ffmpeg -metadata … -codec copy` so no pixels are re-encoded — only the container's global metadata is rewritten. Players and `ffprobe` then surface `title`/`artist`/`comment` directly. This carrier is invisible and lossless to the picture, but it lives in the container and is the easiest to strip with a re-mux.

Steganographic signed sidecars. `export/stego.py` LSB-embeds a *signed* provenance record into the least-significant bit of each RGB channel of an image. The record (`build_provenance`) carries `tool`, `version`, `title`, `author`, scene and chunk counts, and a `content_sha256` digest of the demo. The pipeline writes it into lossless PNG sidecars — `output/provenance/poster_signed.png` plus the transmission-bookend frames — accompanied by a plain-text `provenance.json`. The `democreate stego <png>` command extracts the payload, and with `--demo d.json` it verifies it.

11.2 Honest Survivability: Why the Video Cannot Carry the Hidden Payload

It would be tempting to claim the *video* carries a hidden watermark; it does not, and the manuscript is explicit about why. LSB pixel steganography survives only in a lossless container. An H.264 encode — like any lossy codec — re-quantizes the pixels and destroys the embedded bits completely. The hidden payload therefore lives in the **lossless PNG sidecars only**; the **MP4 itself carries provenance through its container tags and its on-screen bars**, not through hidden pixels. This is the division of labor fig. 12 makes plain: the visible bars and the container tags ride the distributed `.mp4`, while the signed digest rides the PNG poster and bookends that ship beside it. Conflating the two would be exactly the kind of unfalsifiable provenance claim the rest of the manuscript avoids; we state the boundary instead of papering over it. The LSB scheme itself is a textbook spatial-domain embedding (Johnson and Jajodia 1998): one bit per channel, a four-byte length header, and a row-major bit order — chosen for legibility and tamper-evidence, not for robustness against re-compression, which the lossless-PNG carrier deliberately sidesteps.

11.3 A Content Digest That Excludes Render State

The signature is only useful if it is *stable* — if it verifies `True` for the demo it describes and `False` the moment that demo is meaningfully edited. The subtlety is that rendering *mutates* the demo: it fills in audio paths, synced millisecond timestamps, and per-clip durations. Hashing the full serialization would make a freshly-authored demo fail to verify against the one embedded at render time, defeating the purpose. So `_content_digest` hashes only the *authored content* — title, geometry, and the scene/chunk/action structure with narration text and action params — explicitly excluding `audio_path`, `start_ms`, `timestamp_ms`, and `duration_ms`. The result is a digest that is invariant to render state but sensitive to authored change. `verify_provenance` recomputes this digest from a candidate demo and compares it to the `content_sha256` embedded in the PNG: it returns `True` against the original demo and `False` against an edited one (or a missing/corrupt payload). The pairing is therefore tamper-evident — mutate either the demo's content or the image's pixels and verification breaks — while remaining robust to the incidental timestamp churn of a re-render. It is provenance, not encryption: the payload is plain JSON anyone with the module can read.

11.4 Resolution and Quality for Distribution

Distribution-grade output also means distribution-grade pixels. The renderer exposes named 16:9 resolution tiers in `RESOLUTIONS` — `720p` (1280×720), `1080p` (1920×1080), `1440p` (2560×1440), and `2160p`/`4k` (3840×2160) — selected with `RenderConfig.set_resolution` or the `render --resolution` flag. Because every font size and layout metric in the renderer is a fraction of the frame height (sec. 6), a higher tier is *genuinely* higher resolution rather than an upscale: the same `Demo` recomposes crisply at 4K, which we confirm — a 1440p render produces a true 2560×1440 stream, and the 2160p tier a 3840×2160 one (sec. 8). Quality is governed by `VideoConfig.crf` (default **18**, near-visually-lossless — well below x264's default of ~23) and `VideoConfig.preset`, both passed straight to `ffmpeg` by `encode_frame_sequence`/`assemble_video`, so the encode is crisp by default rather than at the codec's lossy baseline.

11.5 The Config Surface

All of the above — resolution and quality, motion, audio, and every provenance field — is reachable from one accessible control surface. `RenderConfig.commented_yaml` (surfaced as `democreate config out.yaml`) emits a fully-commented default YAML in which each commonly-tuned knob carries an inline comment, ready to edit and pass back via `--config`. Since `MetadataConfig` is now a section of `RenderConfig`, a creator's attribution, watermark, and the toggles for the header bar, container tags, and steganography all live in that one file beside the theme, the crf, and the resolution note — making provenance a first-class, declarative part of a render's source rather than an afterthought bolted on at export.

12 References

Astral. 2024. *uv: An Extremely Fast Python Package and Project Manager, Written in Rust*. Https://docs.astral.sh/uv/.

Bain, Max, Jaesung Huh, Tengda Han, and Andrew Zisserman. 2023. "WhisperX: Time-Accurate Speech Transcription of Long-Form Audio." *Proceedings of INTERSPEECH 2023*. https://doi.org/10.21437/Interspeech.2023-78.

Bedos, Nicolas. 2020. *Termtosvg: Record Terminal Sessions as Standalone SVG Animations*. Https://github.com/nbedos/termtosvg.

Brandl, Georg, Matthäus Chajdas, Jean Abou-Samra, and Pygments contributors. 2024. *Pygments: A Generic Syntax Highlighter*. Https://pygments.org.

Brunsfeld, Max, and Tree-sitter contributors. 2024. *Tree-sitter: An Incremental Parsing System for Programming Tools*. Https://tree-sitter.github.io.

Chen, Yanzhe, Kevin Qinghong Lin, and Mike Zheng Shou. 2025. "Code2Video: A Code-Centric Paradigm for Educational Video Generation." *arXiv Preprint arXiv:2510.01174*, ahead of print. https://doi.org/10.48550/arXiv.2510.01174.

Clark, Jeffrey A., and Pillow contributors. 2024. *Pillow: The Friendly PIL Fork (Python Imaging Library)*. Https://python-pillow.org.

CodeVideo. 2024. *CodeVideo: An Event-Sourced Virtual IDE for Generating Coding Tutorials and Videos*. Https://codevideo.io.

European Broadcasting Union. 2020. *EBU R 128: Loudness Normalisation and Permitted Maximum Level of Audio Signals*. European Broadcasting Union. https://tech.ebu.ch/publications/r128.

FFmpeg Developers. 2024a. *FFmpeg Loudnorm Filter: EBU R128 Loudness Normalization*. Https://ffmpeg.org/ffmpeg-filters.html#loudnorm.

FFmpeg Developers. 2024b. *FFmpeg: A Complete, Cross-Platform Solution to Record, Convert and Stream Audio and Video*. Https://ffmpeg.org.

Fowler, Martin. 2005. *Event Sourcing*. Martinfowler.com (Patterns of Enterprise Application Architecture). https://martinfowler.com/eaaDev/EventSourcing.html.

Hexgrad, and Kokoro contributors. 2025. *Kokoro: An Open-Weight, Lightweight Text-to-Speech Model*. Https://huggingface.co/hexgrad/Kokoro-82M.

Johnson, Neil F., and Sushil Jajodia. 1998. "Exploring Steganography: Seeing the Unseen." *Computer* 31

(2): 26–34. https://doi.org/10.1109/MC.1998.4655281.

Kulik, Marcin, and asciinema contributors. 2024. *Asciinema: Record and Share Terminal Sessions*. Https://asciinema.org.

McGugan, Will, and Rich contributors. 2024. *Rich: A Python Library for Rich Text and Beautiful Formatting in the Terminal*. Https://github.com/Textualize/rich.

Meszaros, Gerard. 2007. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Microsoft. 2024. *Playwright: Cross-Browser Automation and Testing Library*. Https://playwright.dev.

Palmér, Moses, and pynput contributors. 2024. *pynput: Control and Monitor Input Devices from Python*. Https://github.com/moses-palmer/pynput.

Pimentel, João Felipe, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. "A Large-Scale Study about Quality and Reproducibility of Jupyter Notebooks." *Proceedings of the 16th International Conference on Mining Software Repositories (MSR)*, 507–17. https://doi.org/10.1109/MSR.2019.00077.

pytest-dev team. 2024. *Pytest: Full-Featured Python Testing Framework*. https://docs.pytest.org/.

Radford, Alec, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. "Robust Speech Recognition via Large-Scale Weak Supervision." *Proceedings of the 40th International Conference on Machine Learning (ICML)*, ahead of print. https://doi.org/10.48550/arXiv.2212.04356.

Ramírez, Sebastián, and Typer contributors. 2024. *Typer: Build Great CLIs, Easy to Code, Based on Python Type Hints*. Https://typer.tiangolo.com.

Recordly. 2024. *Recordly: Browser-Based Screen and Demo Recording*. Commercial screen-recording product.

Resemble AI. 2025. *Chatterbox: Open-Source Text-to-Speech with Emotion and Voice Cloning*. Https://github.com/resemble-ai/chatterbox.

Schoentgen, Mickaël, and python-mss contributors. 2024. *python-mss: An Ultra-Fast Cross-Platform Multiple Screenshots Module in Pure Python*. Https://github.com/BoboTiG/python-mss.

Sweigart, Al, and PyAutoGUI contributors. 2024. *PyAutoGUI: Cross-Platform GUI Automation for Human Beings*. Https://github.com/asweigart/pyautogui.

The Manim Community Developers. 2024. *Manim: A Community-Maintained Python Framework for Mathematical and Explanatory Animations*. Https://www.manim.community.

The Poppler Developers. 2024. *Poppler: A PDF Rendering Library and Command-Line Utilities*. Https://poppler.freedesktop.org.

VSpeak. 2024. *VSpeak: Chunk- and Trigger-Based Narration Synchronization for Generated Walkthroughs*. Software project documentation.

Wood, J., and code-video-generator contributors. 2023. *Code-Video-Generator: Create Code Walkthrough Videos with Manim*. Https://github.com/sleepylemur/code-video-generator.

Zhu, Zeyu, Kevin Qinghong Lin, and Mike Zheng Shou. 2025. *Paper2Video: Automatic Video Generation from Scientific Papers*. arXiv preprint arXiv:2510.05096. https://doi.org/10.48550/arXiv.2510.05096.

Zulko, and MoviePy contributors. 2024. *MoviePy: Video Editing with Python.* Https://zulko.github.io/moviepy/.