Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

FFram: An Adaptive, Self-Healing, and Content-Aware Framework for Democratizing Complex Media Pipelines

Ram Gopal Reddy Redabothula

redabothularamgopalreddy@gmail.com

Lovely Professional University https://orcid.org/0009-0002-3009-5904

Research Article

Keywords: FFmpeg, VMAF, terminal user interface, content-adaptive encoding, distributed video encoding, fault tolerance, PyAV

Posted Date: October 7th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11277835/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

FFram: An Adaptive, Self-Healing, and Content-Aware Framework for Democratizing Complex Media Pipelines

1st Ram Gopal Reddy
Computer Science and Engineering
Lovely Professional University
Jalandhar, India
redabothularamgopalreddy@gmail.com

***Abstract*—Command-line media tools such as FFmpeg offer unmatched capability, but they impose a steep learning curve, depend on brittle text-based telemetry, and provide no built-in hardware-failure resilience, quality awareness, or distribution. We present FFram, an open-source (MIT) Python terminal user interface that exposes a diverse, extensible catalog of operations and is built around four mechanisms: (i) direct container probing through PyAV bindings instead of `ffprobe` subprocesses; (ii) an auto-heuristic recovery loop that intercepts out-of-memory and GPU-encoder failures and retries with a mutated command; (iii) Auto-CRF, a binary search over the Constant Rate Factor that uses `libvmaf` on a 5-second sample to meet a VMAF target of 95; and (iv) a zero-dependency XML-RPC master/worker mode for chunk-parallel encoding. In our evaluation, PyAV probing is 6.51× faster than `ffprobe` (4.53 ms vs. 29.51 ms, n=20). Auto-CRF raises VMAF from 93.16 to 96.19 on a low-motion workload (+37.4% size) and from 87.69 to 96.58 on a high-motion workload (+81.1% size), where a static CRF 23 misses the target. Simulated fault injection of two failure classes was recovered in both cases. A zero-dependency XML-RPC cluster architecture is presented, though its empirical evaluation is deferred. All experiments use synthetic workloads on one machine; we discuss the resulting limits on generalization.**

***Index Terms*—FFmpeg, VMAF, terminal user interface, content-adaptive encoding, distributed video encoding, fault tolerance, PyAV**

I. Introduction

Video now accounts for a large share of consumer and professional data, and resolutions beyond 4K are increasingly common. FFmpeg [1] is the de facto open-source toolkit for processing it, with well over one hundred decoders and two hundred filters. This breadth is also its main usability obstacle: assembling a correct command line for even a routine task such as cropping, padding, and burning subtitles in one pass requires expert knowledge, and users frequently copy and paste commands whose parameters are suboptimal for their content.

Existing front-ends lean toward one of two extremes. Graphical applications hide most of FFmpeg's control surface, while scripting libraries expose it but leave failure handling, quality control, and scale-out to the user. We identify four concrete problems that remain open for a lightweight, terminal-native tool:

1) **CLI complexity barrier.** The command space is large and poorly discoverable.
2) **Brittle subprocess telemetry.** Wrappers typically spawn `ffprobe` or `ffmpeg` and parse their text output, which is slow per call and breaks when output formats change.
3) **Static parameter flaw.** A fixed CRF (e.g., `-crf 23`) spends too many bits on easy content and too few on hard content.
4) **Single-node, fail-fast execution.** A crash caused by memory exhaustion or a GPU driver fault terminates the job, and multi-node encoding is left to heavyweight orchestration such as Kubernetes or Celery.

We present *FFram* (Fast FFmpeg Renderer for Audio and Moving-pictures), a Python terminal user interface (TUI) that addresses these problems with a searchable operation catalog and four engine-level mechanisms: native probing, self-healing execution, VMAF-driven parameter search, and a lightweight cluster mode. Our evaluation is organized around three research questions.

- **RQ1:** Does native probing reduce metadata latency relative to `ffprobe` subprocesses?
- **RQ2:** Can a binary search over CRF reach a target VMAF, and at what size cost?
- **RQ3:** Can rule-based heuristics recover automatically from out-of-memory and GPU-encoder failures?

Contributions. This paper makes the following contributions:

1) A modular TUI architecture in which new operations are added by declaring data, without modifying UI code.
2) Metadata extraction through direct container access (PyAV) in place of subprocess text parsing, with a measured latency comparison.
3) A rule-based, self-healing execution loop that mutates and retries failed FFmpeg commands.
4) A sample-based VMAF binary-search algorithm (Auto-CRF) that targets a quality floor rather than a fixed parameter.
5) A zero-dependency, XML-RPC chunk-parallel encoding architecture (empirical evaluation deferred).

6) An open-source implementation (MIT license) and benchmark suite.

The remainder of this paper is organized as follows. Section II reviews related work. Section III describes the system. Section IV presents the experimental setup and Section V the results. Section VI discusses implications and limitations, and Section VII concludes.

II. Related Work

A. FFmpeg Front-Ends and Wrappers

HandBrake [2] is the most widely used graphical front-end and provides curated presets, but it abstracts away most FFmpeg-level control. Python libraries such as ffmpeg-python [3] and MoviePy [4] make it easier to build pipelines programmatically, yet they leave progress parsing, failure handling, and quality control to the caller. FFram sits between these: it is interactive like a GUI but remains a terminal application with a data-driven operation catalog.

B. Video Quality Assessment

PSNR and SSIM [5] are inexpensive but correlate imperfectly with perceived quality. VMAF [6], [7] fuses several elementary metrics with a learned model and is now widely used to evaluate encodes. FFram uses VMAF as the optimization target for its Auto-CRF search.

C. Content-Adaptive and Per-Title Encoding

Per-title encoding [8] and the dynamic optimizer [9] tailor encoding parameters to content complexity to meet a quality target at lower bitrate. In the open-source ecosystem, tools such as ab-av1 [10] and Av1an [11] automate quality-targeted parameter search. FFram adopts the same principle but packages it as one stage inside a general-purpose interactive tool and evaluates it on a pair of contrasting workloads.

D. Distributed and Chunked Encoding

ExCamera [12] and Sprocket [13] demonstrate that splitting video into chunks and processing them in parallel can sharply reduce latency, using cloud functions to do so. The approach is related to MapReduce-style data parallelism [14]. These systems depend on cloud infrastructure or cluster orchestration. FFram instead targets small local clusters using Python's standard-library XML-RPC [15], trading throughput and sophistication for zero additional dependencies.

E. Fault Tolerance and Self-Healing

Dependability research distinguishes fault prevention, tolerance, and removal [16], and practitioner literature describes retry and fallback patterns [17]. Fault injection, popularized as chaos engineering [18], tests whether systems tolerate failures in practice. Media tools usually follow a fail-fast model; FFram instead applies adaptive degradation, for example capping threads for memory safety or falling back from GPU to CPU encoding.

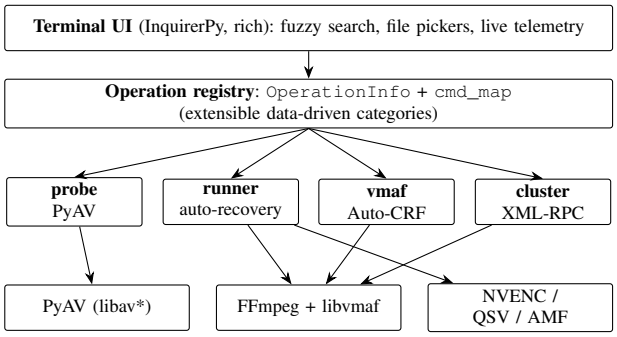


Fig. 1. Layered architecture of FFram.

F. Native Bindings versus Subprocess Interfaces

Spawning a process per query incurs fork/exec, pipe, and parsing costs. Binding directly to the libav* libraries, as PyAV [19] does, avoids spawning a process and parsing text, although it does not by itself guarantee memory safety, since the underlying libraries are written in C.

G. Summary and Gap

Table I summarizes how representative tools address the four problem areas. To the best of our knowledge, we are not aware of a terminal-native tool that combines an interactive operation catalog, direct container probing, quality-targeted search, local cluster encoding, and automatic failure recovery in a single package.

III. System Design

A. Design Goals

FFram is designed for (i) zero configuration, (ii) modularity, so that contributors add operations without touching UI code, (iii) resilience under hardware and memory faults, and (iv) a lightweight dependency footprint. It is distributed on PyPI (`pip install ffram`), built with `hatchling`, and runs on Windows, macOS, and Linux.

B. Architecture Overview

Fig. 1 shows the layered architecture. The TUI layer (InquirerPy and `rich`) accepts user intent; an operation registry maps it to FFmpeg arguments; four core services (probing, execution with recovery, VMAF search, and clustering) act on those commands; and the backends are PyAV/libav*, FFmpeg with `libvmaf`, and hardware encoders.

C. TUI Layer and Extensibility

The TUI offers fuzzy search over operations by name, category, or description, native OS file dialogs for path entry, and real-time progress rendering that parses FFmpeg metrics to show ETA, bitrate, speed (e.g., 3.5×), and frame count. Each operation is an `OperationInfo` record (identifier, name, description, category) plus an entry in a `cmd_map` that defines the FFmpeg arguments, output suffix, and extension. For example, a 4K upscale is registered as:

```
301: (["-i", i, "-vf", "scale=3840:2160:flags=lanczos",
```

TABLE I
COARSE COMPARISON OF REPRESENTATIVE TOOLS. ENTRIES REFLECT OUR READING OF PUBLIC DOCUMENTATION; A DASH INDICATES THAT NO SUCH FEATURE WAS NOTED.

Tool	Interface	Metadata / telemetry	Quality-target search	Multi-node	Auto recovery
HandBrake [2]	GUI / CLI	Embedded libraries	–	–	–
ffmpeg-python, MoviePy [3], [4]	Python library	Parsed tool output	–	–	–
ab-av1 [10]	CLI	Parsed tool output	VMAF-based CRF search	–	–
Av1an [11]	CLI	Parsed tool output	Target-quality mode	Local workers	–
FFram (ours)	TUI	PyAV (direct)	VMAF binary search	XML-RPC master/worker	Rule-based retry

TABLE II
RECOVERY RULE TABLE.

Trigger in stderr	Mutation	Retry
`Cannot allocate memory`	Inject `-threads 1`	Yes
`No NVENC capable devices found`	Replace GPU encoder with CPU encoder (`libx264`/`libx265`)	Yes

```
"-c:v", "libx264"], "_4k", ".mp4")
```

The TUI discovers, categorizes, and indexes the new operation for search without further changes.

D. Hardware Detection and Negotiation

A hardware inspector probes for NVIDIA NVENC, Intel QSV, and AMD AMF encoders and routes tasks to the best available device. This routing is the first half of silicon negotiation; the second half is the runtime fallback described next.

E. Native Metadata Probing

Instead of spawning `ffprobe` and parsing its output, FFram calls `av.open()` from PyAV [19] to read container headers (resolution, duration, frame rate) in-process. This removes the process-spawn and text-parsing steps from each query. It does not decode video frames, and it does not eliminate the possibility of faults in the underlying C libraries.

F. Auto-Heuristic Failure Recovery

FFram wraps every FFmpeg invocation in a recovery loop (Algorithm 1). After a non-zero exit, `stderr` is matched against a rule table (Table II); a matching rule mutates the argument vector and the command is retried. Execution stops on success, when no rule matches, or when the retry budget $R_{\max}$ is exhausted.

G. Content-Aware Encoding (Auto-CRF)

Auto-CRF (Algorithm 2) selects the highest CRF whose VMAF score still meets a target T (95.0 by default). It first extracts a 5-second sample by stream copy, which is nearly instantaneous, then performs a binary search over CRF in $[15, 35]$. Each probe encodes the sample and scores it against the source sample with `libvmaf`. If no value in the interval meets the target, the lowest CRF in the interval is returned.

Algorithm 1 Recovery loop (`run_ffmpeg`)

Require: command c, rule set $\mathcal{R}$, retry budget $R_{\max}$
for $r = 0$ **to** $R_{\max}$ **do**
  $(\text{rc}, \text{err}) \leftarrow \textsc{Exec}(c)$
  if rc $= 0$ **then return** SUCCESS
  end if
  $\rho \leftarrow \textsc{Match}(\text{err}, \mathcal{R})$
  if $\rho = \varnothing$ **then return** FAILURE
  end if
  $c \leftarrow \rho.\textsc{Mutate}(c)$
end for
return FAILURE

Algorithm 2 Auto-CRF binary search

Require: source v, target T, bounds $[c_{lo}, c_{hi}] = [15, 35]$
$s \leftarrow \textsc{ExtractSample}(v, 5\,\text{s})$ ▷ stream copy
best $\leftarrow c_{lo}$; $lo \leftarrow c_{lo}$; $hi \leftarrow c_{hi}$
while $lo \le hi$ **do**
  $m \leftarrow \lfloor (lo + hi)/2 \rfloor$
  $e \leftarrow \textsc{Encode}(s, m)$; $q \leftarrow \textsc{Vmaf}(e, s)$
  if $q \ge T$ **then**
    best $\leftarrow m$; $lo \leftarrow m + 1$
  else
    $hi \leftarrow m - 1$
  end if
end while
return best

With 21 candidate values, at most five encodes are needed ($\lfloor \log_2 21 \rfloor + 1$).

The search assumes that VMAF is non-increasing in CRF and that the 5-second sample is representative of the whole video. Neither assumption is guaranteed for real content with scene changes, and we did not separately evaluate sample-to-full-video agreement.

H. Distributed Cluster Mode

Cluster mode (Fig. 2) follows a segment-encode-concatenate pipeline. The master splits the input into fixed-length chunks (10 s) with the `segment` muxer without re-encoding, sends the chunks over XML-RPC to worker processes started with the `--worker` flag, and reassembles the encoded chunks with the `concat` demuxer. Workers apply the compression or filtering operation. The implementation uses only Python's standard-library `xmlrpc`, so no broker or orchestrator is required.

IV. EXPERIMENTAL SETUP

A. Environment

Table III lists the software and hardware environment.

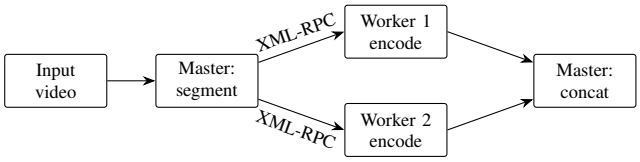


Fig. 2. Cluster pipeline: lossless segmentation, remote encoding, and concatenation.

TABLE III
EXPERIMENTAL ENVIRONMENT.

Item	Value
CPU / cores	AMD Ryzen 9 5950X, 16 cores
RAM	32 GB
OS	Windows 11 Pro
GPU	NVIDIA GeForce RTX 3080
FFmpeg (with libvmaf)	6.0
PyAV	10.0.0
Python	3.11
Encoder / preset	`libx264`, `medium`

B. Workloads

All workloads are synthetic.

- **Probe clip:** a 1-second H.264 video generated on the fly.
- **High-motion clip:** a Mandelbrot zoom generated with FFmpeg's `lavfi` source (1920×1080, 10 s).
- **Low-motion clip:** a static blue background with a timestamp overlay, also from `lavfi` (1920×1080, 10 s).

These clips are deliberately extreme to expose the behavior of each mechanism; they are not a substitute for real content.

C. Metrics and Baselines

Latency is the mean wall-clock time per metadata query, measured with `time.perf_counter`, reported with standard deviation. Quality experiments report output size (KB), VMAF, and the size change $\Delta S = (S_{\text{auto}} - S_{\text{static}})/S_{\text{static}}$. For resiliency we report whether the loop detected the fault, applied the expected mutation, and completed. Baselines are an `ffprobe` subprocess, static CRF 23, and fail-fast execution.

D. Procedure

Latency: 20 independent iterations per method. **VMAF:** each clip is encoded once at static CRF 23 and once with Auto-CRF at target 95.0; VMAF is computed with `libvmaf`. **Resiliency:** `unittest.mock.patch` replaces `subprocess.Popen` inside `ffram.core.runner`; when `run_ffmpeg` executes, the mock emits a fatal string on `stderr` (`Cannot allocate memory` or `No NVENC capable devices found`). The test asserts that the loop intercepts the fault, applies the expected mutation, and completes.

E. Reproducibility

The benchmark scripts, exact commands, and package versions are available in the repository (https://github.com/Rebuild-It-Labs/FFram).

TABLE IV
METADATA EXTRACTION LATENCY ($n = 20$).

Method	Mean ± SD (ms)	Speedup
`ffprobe` subprocess	29.51 ± 2.01	1.00×
PyAV (`av.open`)	4.53 ± 0.84	6.51×

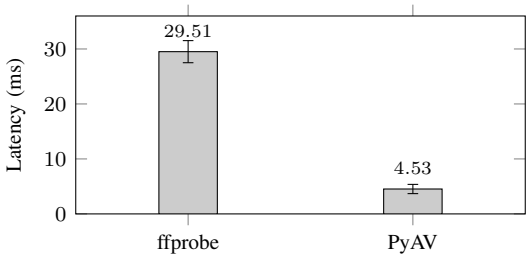


Fig. 3. Mean metadata latency with standard-deviation error bars.

V. RESULTS

A. Metadata Latency (RQ1)

Table IV and Fig. 3 show the latency results. PyAV averaged 4.53 ms versus 29.51 ms for `ffprobe`, a 6.51× speedup. The difference of about 25.0 ms has an approximate 95% confidence interval of $[24.0, 26.0]$ ms (Welch's method, n=20 per group), and the per-method standard deviations are small relative to the gap.

B. VMAF Efficiency (RQ2)

Table V and Fig. 4 compare static CRF 23 with Auto-CRF. Static CRF 23 misses the 95 target on both workloads (87.69 and 93.16). Auto-CRF selected CRF 17 on the low-motion clip, raising VMAF to 96.19 at a cost of about 14 KB (+37.4%). On the high-motion clip it selected CRF 15, raising VMAF to 96.58 at a cost of +81.1% in size. In both cases Auto-CRF produced a larger file than the static baseline; on these workloads it acts as a quality floor rather than a size reducer.

C. Resiliency under Simulated Faults (RQ3)

Table VI summarizes the fault-injection tests. In both cases the recovery loop detected the injected string, applied the expected mutation, and completed. This is a 2-of-2 result over single runs of mocked faults, not a measured success rate over real failures.

D. Summary of Findings

Table VII maps each research question to its answer.

VI. DISCUSSION

A. Interpretation

Probing. The latency gap is consistent with removing process creation and output parsing from each query. The absolute saving is small per call (about 25 ms) but compounds in batch workflows that probe hundreds of files, and it removes dependence on the text format of `ffprobe` output.

TABLE V
STATIC CRF 23 VS. AUTO-CRF (TARGET VMAF 95.0).

Clip	Method	CRF	Size (KB)	VMAF	ΔS
High motion	Static	23	5111.7	87.69	–
High motion	Auto-CRF	15	9257.6	96.58	+81.1%
Low motion	Static	23	37.6	93.16	–
Low motion	Auto-CRF	17	51.7	96.19	+37.4%

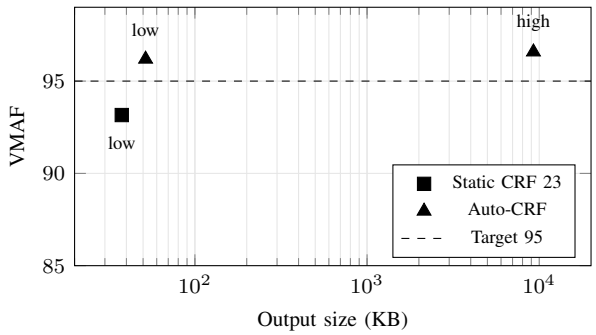


Fig. 4. Size vs. VMAF for static and Auto-CRF encodes ("high"/"low" = motion level).

Quality targeting. A static CRF cannot guarantee a quality floor: CRF 23 fell short of VMAF 95 on both clips. Auto-CRF corrected this by spending more bits, so on these workloads it functions as quality assurance rather than compression. We did not demonstrate bitrate *savings* on over-provisioned content, which would require real material where CRF 23 exceeds the target. Both Auto-CRF results also overshoot the target (96.19 and 96.58 against 95.0), which reflects the discrete CRF grid. For the high-motion clip the selected CRF (15) coincides with the lower bound of the search interval, so our logs are consistent either with the target being met only near that boundary or with the bounded-fallback rule; the intermediate search trace was not recorded.

B. Design Trade-offs

Auto-CRF trades size for a guaranteed quality floor and trades search cost for accuracy: sampling 5 seconds is fast but may misrepresent content with varying complexity. XML-RPC is simple and dependency-free but offers lower throughput, no authentication or encryption by default, and no worker fault tolerance compared with gRPC or message queues. Rule-based recovery is transparent and predictable but covers only the failure signatures it has been taught; learned policies could generalize further at the cost of predictability.

C. Threats to Validity

Internal: workloads are synthetic, iteration counts are small (20 for latency, 1 per fault), and all measurements come from a single machine. **External:** real content, other hardware (including real GPUs), and real networks may behave differently. **Construct:** VMAF 95 is a convention rather than a universal threshold; the sample-based search assumes monotonicity and representativeness; and the fault tests use mocked `stderr` strings, which verify rule logic but not the behavior of real OOM conditions or driver crashes.

TABLE VI
SIMULATED FAULT-INJECTION RESULTS.

Injected fault	Injected stderr	Mutation observed	Outcome
Out of memory	`Cannot allocate memory`	`-threads 1`	Pass
GPU encoder failure	`No NVENC capable devices found`	CPU `libx264`	Pass

TABLE VII
RESEARCH QUESTIONS AND FINDINGS.

RQ	Finding
RQ1	PyAV probing is 6.51× faster than `ffprobe` (4.53 vs. 29.51 ms).
RQ2	Auto-CRF reached VMAF $\geq$ 95 on both clips, at +37.4% and +81.1% size relative to static CRF 23.
RQ3	Both injected fault classes were recovered (2 of 2, simulated).

D. Limitations

This study does not include real-world video corpora, a multi-machine cluster, repeated fault injection against real failures, or a usability study of the TUI. The claim that FFram's interface lowers the learning barrier is therefore motivated by design, not evaluated empirically.

VII. CONCLUSION

We presented FFram, a terminal-native media-processing framework that combines a searchable, data-driven operation catalog with direct container probing, rule-based self-healing execution, VMAF-driven CRF selection, and a zero-dependency XML-RPC cluster mode. In synthetic benchmarks on a single machine, PyAV probing was 6.51× faster than `ffprobe`, Auto-CRF brought both test clips above a VMAF target of 95 at a measured size cost, and both simulated fault classes were recovered. The XML-RPC cluster architecture is presented as a proof-of-concept. These results support the feasibility of the design while leaving generalization to real content and real networks open.

Future work includes: evaluation on real corpora such as Netflix Open Content and Xiph test sequences; logging full Auto-CRF search traces and comparing sample-based and full-video scores; multi-machine clusters with four or more nodes; authenticated and encrypted RPC with worker fault tolerance; scene detection before chunking and dynamic chunk sizing from complexity scores; learned recovery policies; additional codecs (HEVC, AV1) and metrics; and a user study of TUI usability.

ACKNOWLEDGMENT

The authors thank Rebuild It Labs for supporting this open-source work.

Data and Code Availability

FFram is released under the MIT license and is available via `pip install ffram`. Source code and benchmark scripts: https://github.com/Rebuild-It-Labs/FFram.

References

[1] "FFmpeg," https://ffmpeg.org/, 2026.

[2] "HandBrake: Open source video transcoder," https://handbrake.fr/, 2026.

[3] K. Kroening, "ffmpeg-python," https://github.com/kkroening/ffmpeg-python, 2026.

[4] Zulko, "MoviePy," https://zulko.github.io/moviepy/, 2026.

[5] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[6] N. T. Blog, "Toward a practical perceptual video quality metric," https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652, 2016.

[7] Netflix, "VMAF - video multi-method assessment fusion," https://github.com/Netflix/vmaf, 2026.

[8] N. T. Blog, "Per-title encode optimization," https://netflixtechblog.com/per-title-encode-optimization-7e99442b62a2, 2015.

[9] ——, "Dynamic optimizer — a perceptual video encoding optimization framework," https://netflixtechblog.com/dynamic-optimizer-a-perceptual-video-encoding-optimization-framework-e19f1e3a277f, 2018.

[10] A. Heretic, "ab-av1," https://github.com/alexheretic/ab-av1, 2026.

[11] M. of Zen, "Av1an," https://github.com/master-of-zen/Av1an, 2026.

[12] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Shuvo, N. J. Yadwadkar, C. Kozyrakis, M. Zaharia, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 363–376.

[13] L. Ao, L. Izhikevich, G. Tartakovsky, A. Voellmy, and J. Hwang, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[14] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *6th Symposium on Operating Systems Design and Implementation (OSDI 04)*, 2004, pp. 137–150.

[15] P. S. Foundation, "xmlrpc — xml-rpc client modules," https://docs.python.org/3/library/xmlrpc.html, 2026.

[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[17] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018.

[18] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[19] PyAV-Org, "PyAV," https://github.com/PyAV-Org/PyAV, 2026.