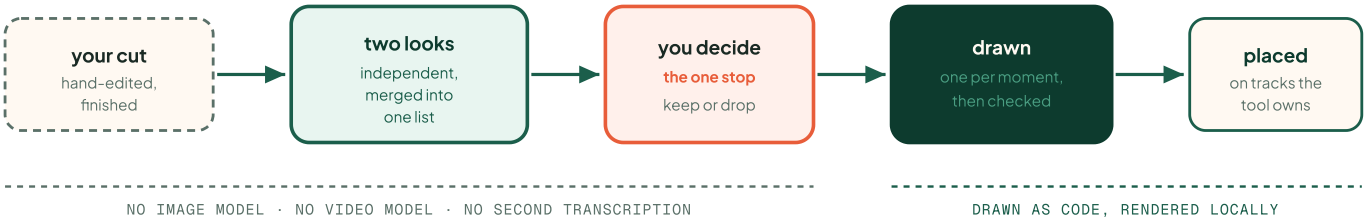
The CapCut Auto-Overlay Blueprint

The architecture and the data flow behind a tool that finds the moments in your cut, draws a graphic for each one, and places it on the timeline with sound.

Ben Lim · BITSTAQ

benlimjw@bitstaq.com · bitstaq.com



What this is. The architecture and the data flow: what arrives at each stage, what leaves it, and why each boundary sits where it does.

What it is not. Source code, commands, the decision rules the tool judges moments by, or the drawing instructions it hands its agents. Section 8 states exactly where that line falls and why it sits there rather than somewhere else.

Version 1 · 19 August 2026 · bitstaq.com/resources/capcut-auto-overlay

Written from the tool as built, and from the workflow record of the video that announced it.

Questions, corrections, or a build of your own you want to argue about · **benlimjw@bitstaq.com**.

CONTENTS

What's in here

Eight sections, read as one flow of data. One and two are why the tool exists at all. Three through seven follow a single video through it, in order. Eight is where it still breaks and where the line is drawn.

HOW TO READ THIS

This is a data-flow document, not a build specification. Every section answers the same three questions: what arrives here, what leaves here, and why it's shaped that way. Where a design choice buys something at a cost, both sides are stated. Where the tool is unfinished, it says so rather than describing the version I'd like it to be.

SECTION 01

What this is, and what it runs on

One skill, called once, against a CapCut project you have already finished cutting.

The tool is packaged as a single skill for an AI coding agent. You call it, you name the CapCut project, and it works through the whole run from reading the timeline to placing finished graphics back onto it. There is one stop in the middle where it needs a decision from you. Everything else runs without you.

The output is on-screen motion graphics: the small animated panels, callouts, annotations and diagrams that appear over footage to make a point land. Each one is timed to what is being said underneath it, positioned to avoid whatever on screen matters, and accompanied by a sound as it arrives and another as it leaves.

What it costs to run

Nothing beyond the plan you already have. This is the first question people ask, and the answer is unusual enough to be worth stating precisely, because "AI animation" carries a billing model with it that does not apply here.

> THE LOAD-BEARING FACT
>
> There is no image model and no video model anywhere in this. The graphics are *drawn*: an agent writes the markup, the styling and the timeline for each one as code, and that code is rendered locally into a video file with a transparent background. Generation-priced tools like Higgsfield and Runway bill per generation because a model produces each frame. Here nothing produces frames except your own machine.

The practical consequence is that the expensive resource is your agent subscription's context, not a metered credit balance. Re-drawing a graphic you didn't like costs attention and machine time. It does not cost money per attempt, which changes how freely you are willing to iterate, and that turns out to matter more than the raw price does.

There is also no second transcription bill, which is a design choice rather than a happy accident, and Section 4 is entirely about it.

Where it sits in a larger workflow

This is the second half of a two-step pipeline. An earlier tool cuts the raw recording down: it drops silences, dead air and abandoned takes, and publishes a rough timeline into the editor. You then hand-edit that timeline for as long as you want. This tool runs after that, on whatever you actually ended up with.

The ordering is not a convenience. It is the constraint the entire architecture is built around, and Section 3 explains why.

> ON "FULLY AUTOMATED"
>
> The middle of this pipeline is often drawn as unattended, and it isn't quite. The cutting stage runs an interactive conversation about the content before it publishes anything. This stage stops once for approval. Both stops are short, but a description that leaves them out is describing a tool that doesn't exist yet.

SECTION 02

The bottleneck it removes

Prompt-to-animation was solved before this existed. Prompt fatigue wasn't.

There is already a good answer to "how do I make a motion graphic without learning motion design". Tools like HyperFrames take a plain-language description of a graphic, what it says, how it looks, where it enters from and where it lands, and produce the animation. That capability is the floor this tool is built on, not something it replaces.

So the problem is not drawing. The problem is that a long video needs many graphics, and every one of them needs a prompt, and each prompt needs three things that are not equally easy to supply.

WHAT A PROMPT NEEDS	WHY IT'S EXPENSIVE TO SUPPLY BY HAND	WHAT HAS TO BE TRUE FOR AN AGENT TO SUPPLY IT
Describe	The cheapest of the three. You watched the video, you know what the point was.	It has to know what is being said at that instant, and what the footage is already showing.
Time	Sends you back into the timeline to read exact in and out points off the ruler, per graphic.	It has to read the real timeline, not an approximation of it, and to a resolution finer than a second.
Place	You have to look at the frame underneath and find space that isn't already carrying something.	It has to see the actual frames from that stretch of video, not just read the words spoken over them.

Describing is the part a person is genuinely good at. Timing and placement are clerical, they are unforgiving, and they are the two that make you get up and check. On a short video you absorb that cost. On a long one, the quality of your prompts drops off as you go, which means the back half of your video gets worse graphics than the front half for no reason other than fatigue.

> THE WHOLE DESIGN IN ONE LINE
>
> Automate the clerical two by giving an agent the same two things a human would go and look at: **the real timeline**, and **the real frames**. Everything in Sections 3 through 7 is a consequence of taking that seriously.

What this is not trying to be

- **Not a template filler.** Nothing is selected from a library of pre-made animations and populated with your text. Each graphic is authored for its own moment.
- **Not an editor.** It doesn't cut, reorder, retime or colour your footage. It adds to a timeline you own and it reads everything else.
- **Not a subtitle tool.** Captions are a separate concern with a separate tool, and deliberately so.

SECTION 03

What goes in

Two inputs. One of them is the single most load-bearing decision in the design.

The tool takes exactly two things, and it takes them from work that has already happened rather than asking you to prepare anything.

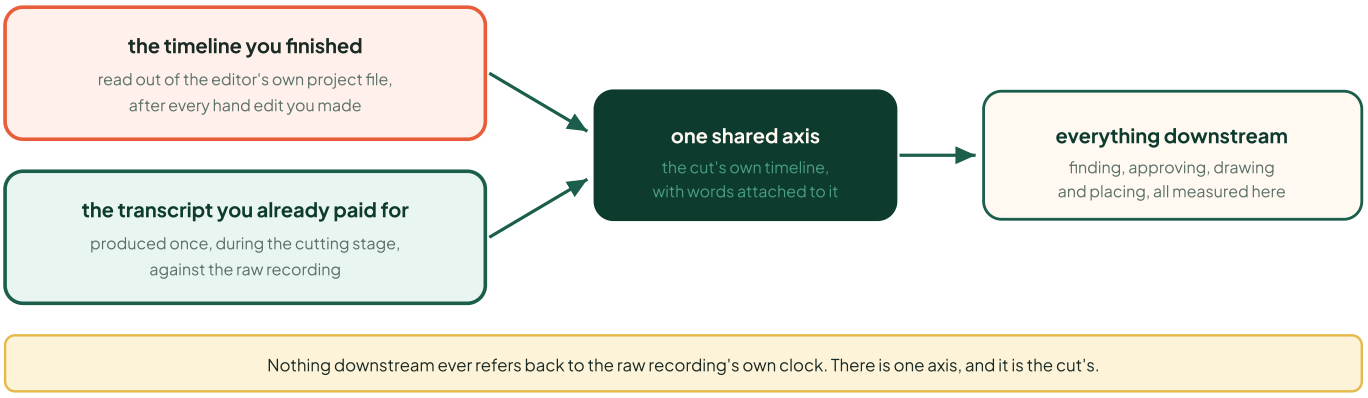


Figure 1. Both inputs are resolved onto a single axis before any decision is made. Getting that right once is what stops every later stage from having to think about it.

Why the finished timeline, and not the tool's own earlier output

The cutting stage that ran before this one produced a timeline of its own. It would have been simpler to keep that around and treat it as the truth. The tool deliberately doesn't. It re-reads the project as it stands, after every manual change you made in the editor.

The reason is that people edit. You will drop a clip that reads badly, tighten a pause, reorder two sections, add a speed ramp. Every one of those moves everything after it. A tool that placed graphics against the pre-edit timeline would be right at the start of the video and progressively wrong toward the end, and the failure would be invisible until you watched it back. Reading the timeline you actually have is what makes graphics land *on* the footage instead of near it.

The measured version of this was visible before the change: registration was fine early in a video and drifted further out the longer it ran. That is the shape of a coordinate error accumulating, and it goes away entirely once the post-edit timeline is the authority.

What that choice costs

It is not free, and the cost falls in an awkward place. Because every approved position is measured against the timeline as it stood at approval, a single trim to the main video track *after* approval invalidates all of them at once. The tool refuses rather than placing graphics it knows are now wrong, and the run restarts from the beginning.

> SAY THIS BEFORE ANYONE STARTS, NOT AFTER
>
> Finish the cut, then approve. One trim after approval costs the whole approved list and every graphic drawn from it. This is the correct behaviour, but it is only tolerable if the person running it knows about it in advance, so any implementation of this design should say it loudly at the start of a run rather than in an error message at the end.

There is a workflow habit that removes most of the pain. Put anything you add later, extra footage or an inserted clip, on a track *above* the main video rather than splicing it into the main track. Splicing shifts everything after the insert. A track above it shifts nothing, because the graphics register against the main track only.

SECTION 04

Re-projected, never re-transcribed

The words are moved onto the cut's axis by arithmetic. Nothing is sent away to be transcribed a second time.

The transcript that arrives here was made against the raw recording, before anything was removed. After the cut, its timings are wrong by a growing amount, because every dropped silence and abandoned take pulls everything after it earlier. The words are still right. Only their positions are stale.

There are two ways to fix that. One is to export the cut, send the new audio away, and transcribe it again. The other is to work out exactly what was removed and shift the words you already have. The tool does the second.

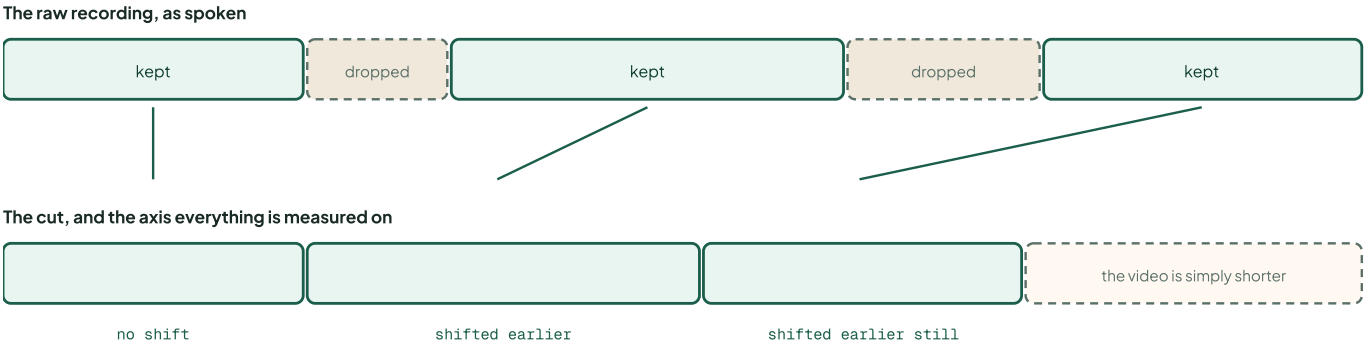


Figure 2. Each surviving stretch carries its own words with it, moved by however much was removed ahead of it. The words never change. Only their coordinates do.

Why this is the interesting choice

Re-transcribing looks easier and it is worse for two reasons, in that order of importance.

Every run would give you slightly different words

Transcription is a model call. Give it the same audio twice and you can get back slightly different word boundaries, or a word split where it wasn't before. Downstream, a moment's start time is derived from where a word begins, so timings that wobble between runs produce graphics that arrive a beat late on one run and on time on the next. That is exactly the kind of bug that is impossible to reproduce and therefore impossible to fix.

Shifting a transcript you already have is arithmetic. Arithmetic returns the same answer every time, so two runs over the same cut produce identical positions. Determinism here isn't a nicety. It is what makes the tool debuggable at all.

You already paid for it

The cutting stage paid for a transcription once and kept the result. Sending the same speech away again is a second bill for a fact you already own. Avoiding it is why there is no metered cost in this stage, which is the claim Section 1 makes and this is where it's earned.

> THE GENERAL PRINCIPLE, IF YOU BUILD SOMETHING LIKE THIS
>
> When a stage upstream of you has already paid a model to establish a fact, treat that fact as an asset and move it forward. Re-deriving it is more expensive and less stable, and the instability is the part that will cost you, not the money.

SECTION 05

Finding the moments

Two independent looks at the same material, merged into one list. Not one look asked twice.

This is the stage that replaces you watching your own video with a notepad. What arrives is the cut and the words now sitting on its axis. What leaves is a list of candidate moments, each one a stretch of the timeline with a start, an end, and a description of what a graphic there would be doing.

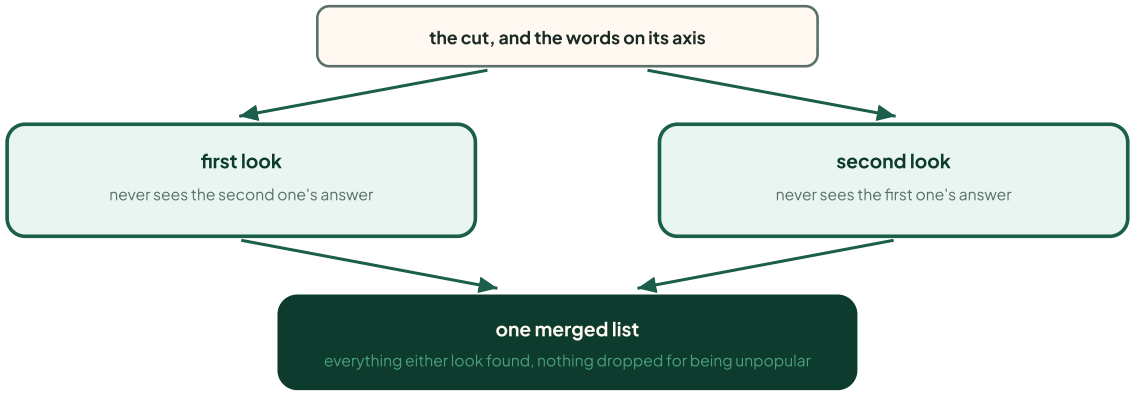


Figure 3. The two looks are dispatched separately and neither is told about the other. Merging keeps the union, not the intersection, because a moment only one pass noticed is still a moment.

Why two looks and not one asked twice

These are not the same thing, and the difference is the whole point of the stage.

Asking a single pass to produce two answers gives you one opinion wearing two hats. It has already anchored on its first read of the material, and the second answer will look a great deal like the first. Two genuinely separate dispatches, each starting cold on the same material, give you two opinions. What you learn from the disagreements is worth more than the agreements, because a moment that only one of them noticed is exactly the moment a single pass would have quietly missed.

An implementation of this should treat the separation as something to enforce rather than to trust. If the run cannot prove that two dispatches actually happened, it should refuse rather than proceed on one answer absorbed twice. That refusal has caught the failure it exists for, which is the only reason to keep a check like it.

Merge by union, not by vote

Overlapping findings are combined. Findings unique to one pass are kept as they are. There is no quorum rule that discards a moment for having been seen once, and that is deliberate: the human review immediately after this stage is cheap, so the correct thing to do is over-produce here and let a person cut. A vote here would be a machine silently making the decision a human is about to make anyway, with less information.

> A DESIGN RULE WORTH STEALING
>
> Put the widest net immediately before the cheapest human decision. Filtering aggressively upstream of a review stage buys nothing, because the reviewer was going to look at the list either way. It only loses you the items that were hardest to find.

What this stage does not do

It doesn't design anything. A finding says where a moment is and what it's about. It says nothing about what the graphic will look like, and no drawing has happened yet. That separation is what makes the approval stop worth having, because you're approving *subject matter*, not artwork, and that is a much faster judgment to make.

SECTION 06

The approval stop

One place where a person decides. It sits before anything is drawn, which is what makes saying no cheap.

The merged list is written out as a sheet you open and read. Each candidate carries a number, where it sits on the timeline, and what a graphic there would be doing. You answer by number: keep it, or drop it. That's the entire interaction.

Why the stop is here and not somewhere else

There is a version of this tool with no stop at all, and a version with the stop at the end, after everything is drawn. Both are worse, and for the same reason.

WHERE THE STOP COULD GO	WHAT IT COSTS TO REJECT SOMETHING THERE
No stop	You find out what it decided by watching the finished video. Every disagreement is now a re-run.
After drawing	Everything you reject has already been drawn, checked and rendered. The work is done and thrown away.
Before drawing (this design)	Rejecting costs one line of reading. Nothing has been drawn, so nothing is wasted.

The general form is that a human gate is worth the most at the point where the work behind it is cheapest and the work ahead of it is expensive. Here, finding is cheap and drawing is expensive, so the gate goes between them.

What leaves this stage

Approval is what turns a candidate list into a fixed one. Three things become true at that instant, and everything downstream depends on all three.

- **The list stops moving.** No moment is added, removed or retimed after this without a deliberate re-open, which is a recorded action rather than a silent one.
- **Every kept moment gets a permanent identifier.** It is minted here, it never changes, and it becomes the name of that graphic's clip in the editor. That's the detail that makes the whole thing workable in practice: when you look at a graphic on your timeline and want it redrawn, the name of the clip is already the thing you say back to the tool. There's no lookup step and no ambiguity about which one you mean.
- **The frames get pulled.** Stills are captured from each approved moment's own stretch of video, and only from approved ones. Judging whether a moment deserves a graphic doesn't need a picture, so pulling frames before approval would be work spent on moments about to be dropped.

> ON IDENTIFIERS AS AN INTERFACE
>
> Giving each moment an identity that survives into the editor's own clip names is a small decision with an outsized payoff. It means the artifact on your timeline and the thing the tool calls it are the same string, so the human and the machine are always talking about the same object. Any system that hands work back to a person inside a third-party application benefits from this.

SECTION 07

Drawing and placing

One graphic per approved moment, drawn from that moment's own brief and its own frames, checked before it goes anywhere, then placed on tracks the tool owns.

What each drawing task is handed

Each approved moment becomes one drawing task, and that task receives two things and nothing else that matters: the brief written for that moment, and the stills captured from that moment's own stretch of video.

The stills are the load-bearing input. They are why the tool can put a graphic in the empty half of the frame instead of over the webcam bubble, and why it annotates the diagram that is actually on screen rather than a generic one. An agent reading only the words spoken at that instant is guessing about the picture. An agent looking at the picture is not.

What comes back is the graphic itself, authored as code: the markup, the styling and the animation timeline for that one moment. Nothing is chosen from a library of pre-made animations and nothing is a template with your text dropped into it.

> WHERE THE BRIEFS ARE WRITTEN MATTERS
>
> The briefs are expanded centrally, before any drawing task is dispatched, rather than by each drawing task for itself. The reason is consistency: separate agents each writing their own brief describe things slightly differently, and slightly different descriptions produce visibly different graphics inside one video. Writing them in one place first is what keeps a video looking like one video.

Batches, and why they exist

Drawing tasks run several at a time rather than one after another. The obvious benefit is wall-clock time. The less obvious one is the real reason: each agent is working within a finite amount of context, and an agent asked to draw too many graphics in one session gets worse toward the end of them. Splitting the work is a quality decision that happens to also be a speed decision.

The gate

Every finished drawing goes through a mechanical gate before it gets anywhere near the timeline. The checks are ordinary structural ones: is this the thing that was asked for, does it stay inside the space it was given, does it respect the areas it was told to avoid.

Most failures are repaired. A repair is scoped to the aspect that failed, not a redraw from scratch, and that distinction is not cosmetic. These are model outputs, so re-running the same request produces a *different* graphic rather than a corrected one. Repairing in place keeps everything that was already right.

A handful of failures deliberately escalate to a person instead of being repaired automatically. Those are the ones where an automated fix would most likely restyle something that was already correct, and the honest answer is that a machine cannot tell the difference. Naming that category up front is more useful than pretending the gate can repair everything.

Rendering, and why it has to be video

The editor works in video clips. It will not take a web page. So every accepted graphic is rendered locally into a video file with a transparent background, at the project's own resolution and aspect ratio, detected rather than assumed.

That detail is easy to skip past and it's the difference between a graphic that lands and one that doesn't. Rendering at the wrong dimensions and letting the editor scale it moves everything inside the frame, which means the empty corner the agent carefully aimed for is no longer where it aimed.

Placing

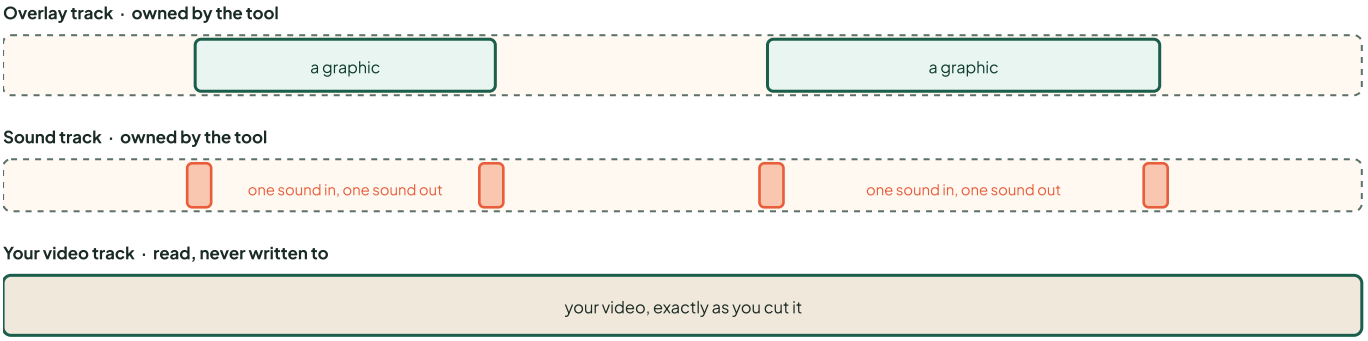


Figure 4. Everything the tool creates lives on tracks it owns. Your own track is read for its timings and is never modified, which is what makes a re-run safe to attempt.

The tracks the tool writes to are its own, and that ownership boundary is what makes the rest of the behaviour possible to reason about. Two rules follow from it.

- **Re-running replaces the family, it doesn't add to it.** A second run clears the graphics and sounds it placed before and puts the current set down. Otherwise every re-run would stack a duplicate on top of the last one, and the timeline would degrade silently with use.
- **A re-run that can't produce everything refuses, rather than shipping less.** If a redraw fails on a re-run, the tool changes nothing at all instead of placing what it has. Because a re-run rebuilds the whole family, a partial write would quietly remove a graphic you had already accepted and watched. Refusing leaves the video you already have playing while you retry.

> THE OTHER SIDE OF THAT OWNERSHIP
>
> Because these tracks belong to the tool, nudging a graphic or a sound by hand in the editor will be lost the next time it rebuilds them. Edits to your own video track are untouched. Any implementation should say this out loud before a rebuild, not after one.

SECTION 08

Limits, and where the line is drawn

What this doesn't do yet, stated plainly, then exactly what this document withholds and why.

This is a first working version. The architecture above is real and it runs, but there are places where the output still needs a person looking at it, and describing them as solved would make this document less useful, not more.

Where it still needs you

Legibility over busy footage

A graphic can be placed correctly, in genuinely unoccupied space, and still be hard to read. Dark, text-heavy footage underneath is the case that breaks it: the placement logic is about *occupancy*, and legibility is about contrast, which is a different question that the gate does not currently ask. This is the limitation most likely to show up on your first run.

Graphics that restate what's already on screen

If the footage underneath already says the thing in writing, a graphic repeating it adds nothing and should have been dropped. The finding stage is drawn toward moments where something is being explained, and a slide explaining it is exactly such a moment, so this is a predictable failure rather than a random one. Catching it is a judgment about redundancy, and for now that judgment is yours, at the approval stop.

The sounds are basic

Every graphic gets the same pop as it arrives and the same pop as it leaves. A panel that slides in from the left should get a sound that slides. The pairing mechanism is right and the library behind it is thin, so this is a matter of widening the library rather than changing the design.

The trim rule is a constraint, not a bug

Section 3 covers it: one trim to the main video track after approval costs the whole approved list. It follows directly from reading the real timeline, which is the choice that makes everything else work, so it isn't going away. Plan around it rather than waiting for it to be fixed.

What this document includes, and what it doesn't

This blueprint shares the architecture and the data flow. It deliberately stops short of the parts that would let you skip the work of building it, and stating that precisely is more useful to you than pretending the line isn't there.

IN HERE: THE SHAPE

- Every stage, and what crosses each boundary between them
- Which inputs are authoritative and why
- Where the human decision sits, and the reasoning for putting it there
- The design choices that cost something, with both sides stated
- The ownership boundary on the timeline, and the two rules that follow from it
- Every limit I know about, named as a limit

NOT IN HERE: THE LEVERAGE

- **No source code** and no commands. Nothing you could paste into a file and run.
- **No decision rules.** What makes a stretch of video worth a graphic, and how a candidate is judged, is not described.
- **No drawing instructions.** The brief format the drawing tasks receive, and the standards they author against, are not here.
- **No gate contents.** That a gate exists and what kind of thing it checks is here. The checks themselves are not.
- **No component library or brand kit.** The curated set the drawings are built from is not described.

The line is drawn differently here than on the subtitle blueprint that preceded it, and deliberately. That one is a complete build specification, because there is nothing about a subtitle segmenter I mind giving away. This one is the tool I'd most like to keep improving, so the reasoning is public and the rules are not. If you'd rather have the reasoning and build your own rules, that is a completely legitimate outcome, and it's why the reasoning is this detailed.

How to read the claims in here

Everything above is written against the tool as it stands in August 2026, and against the workflow record of the video it was built for. Where something was observed rather than designed, this document says so in words rather than in numbers, because a number quoted out of one run is a fact about that run and reads like a fact about the tool.

If you'd rather not build it

This is the first working version and it's wired up on my own machine. I haven't packaged it for anyone else. If you want that to happen, say so at **benlimjw@bitstaq.com** or from the page this came from, and if enough people do I'll put the time in. If you build your own from this instead, that's a completely legitimate outcome.

Ben Lim · benlimjw@bitstaq.com

BITSTAQ · bitstaq.com/resources/capcut-auto-overlay · Version 1, 19 August 2026