Episode Book

EPISODE #5465

Packaging AI Pipelines So They Actually Get Reused

PUBLISHED	22 September 2026 at 14:06 UTC
DURATION	24:30
LISTEN	myweirdprompts.com/episode/ai-pipeline-packaging-recipes/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.2
BOOK GENERATED	19 October 2026 at 17:36 UTC

SYNOPSIS

Daniel has a working podcast pipeline: a language model writes scripts, retrieval tools feed it, and text-to-speech reads it out. But the occasional Hebrew word keeps breaking things, and the pipeline itself is trapped in his head. This episode digs into two questions at once — how to handle Hebrew code-switching in TTS, and how to package a chained AI workflow so it survives past the project that spawned it. We talk through script-based segmentation versus small classifiers, why unvocalized Hebrew makes TTS inherently unstable, the tradeoff between chaining small models and using one large one, and why containerization is the answer to dependency hell. Plus: the "recipe" as a distinct layer between a model and a project, and a pre-flight risk score that flags bad inputs before you burn compute.

TRANSCRIPT



Corn

Here's the thing that keeps nagging at me about Daniel's question before we even get into it. He's got a pipeline that works. The episodes go out. And yet he's asking how to make it reusable, which is a completely different problem from making it work.



Herman

The gap between "it runs on my machine" and "it runs on anyone's machine" is where most of the AI ecosystem quietly dies.



Corn

Right. So Daniel wrote in with a whole thing. He wants our thoughts on best practices for coming up with what people call recipes for using AI tools. His specific case: the podcast pipeline, which chains a language model for scripts, retrieval tools, and text to speech. And the little nuance he's been trying to nail is making sure the occasional Hebrew words actually render in Hebrew. Code-switching, basically. Episodes are in English, but Hebrew words show up.



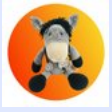
Herman

And he's been down this road before.



Corn

He has. He references a previous conversation where we talked about using a small language model, maybe a classifier, to identify which words are Hebrew, then convert them to Hebrew characters, then add the tags Chatterbox needs. Which means chaining multiple small models, each doing one discrete job, with Python as the glue.



Herman

That's the architecture he's proposing.



Corn

Then the bigger question. When you share something like this in open source, you can share the full project, or the models, or... and here's where he says he's reaching for a word. The word is recipe. And his motivation isn't entirely altruistic, which I appreciate. He wants to package it because it's a pattern that'll be useful for his future projects, and rather than trying to remember how he did it, which models, it could all be documented.



Herman

Executable documentation.



Corn

Then containerization. Could this be a containerized workload for integration into pipelines? On first run the user downloads the model weights, caches them server side. Ship code with replaceable variables and include the models. He says he hasn't really thought about containerization in the context of workflow components before.



Herman

Which is interesting, because that's exactly what it's for.



Corn

He also notes that we previously told him Replicate doesn't support this kind of chaining workflow. And he's noticed people using ComfyUI for non-image workloads, but he feels it's an awkward fit. So the actual question: for people looking to modularize and package their own chained ASR or TTS normalization pipelines, what's the best format and the best distribution channel, whether for your own use, for others, or both?



Herman

That's four or five questions wearing a trench coat.



Corn

It is. So let's unpack it. Start with the core technical challenge, the Hebrew code-switching, then zoom out to packaging and sharing.



Herman

First thing to define, because Daniel's reaching for the word recipe and I think he's right to. A recipe in this context is the orchestration knowledge. Which models, in what order, with what preprocessing. It's distinct from a model and distinct from a project. If you share a model, you've shared a component. If you share a project, you've shared everything including the stuff that only makes sense for your use case. A recipe is the middle layer.



Corn

The thing that gets lost.



Herman

The thing that always gets lost. You come back eighteen months later and you've got a folder called tts_stuff_final_v3 and no idea which Whisper variant you were using or why there's a file called fix_hebrew_again.py.



Corn

I've watched Daniel's repos. That file exists.



Herman

So the use case. Podcast pipeline, English, occasional Hebrew words. The goal is to detect those words, convert them to Hebrew script, add whatever tags the TTS engine needs. And the two threads are: how do you build the recipe for this specific task, and how do you package and share it.



Corn

There's a paper that validates the whole concept, actually. PolyNorm, out of the EMNLP industry track. Their contribution wasn't a model, it was a language-agnostic pipeline for automatic data curation and evaluation. They shipped the recipe.



Herman

The pipeline is the contribution. That's the framing.



Corn

So where do we start with the Hebrew?



Herman

With a question Daniel didn't quite ask, and it changes everything. Are the Hebrew words already in Hebrew script, or are they transliterated into Latin characters?



Corn

Say more.



Herman

If they're already in Hebrew characters, you don't need a classifier. Hebrew characters live in a specific Unicode block. You segment by script, deterministically, with a regex. There's a paper, SFMS-ALR, script-first multilingual synthesis, and that's exactly what it does. It segments input text by Unicode script, then applies adaptive language identification per segment, then generates a unified SSML representation with lang or voice spans, synthesized in a single TTS request. No retraining, works with existing voices.



Corn

So the small classifier Daniel's been planning might be solving a problem he doesn't have.



Herman

If the words are in Hebrew script, yes. The classifier is only needed if the source text has Hebrew words transliterated. Shabbat, mitzvah, tikkun olam, written in Latin letters, and you want them rendered in Hebrew script. Then you need something to detect them, because Unicode segmentation can't help you. They look like English words.



Corn

And that's the harder problem. Because "shabbat" is a string of Latin characters. So is "shallot."



Herman

And a classifier can do it. It's a small, bounded task. But you need training data, and you need to handle the edge cases. Is "amen" Hebrew? It's in English dictionaries. Is "hallelujah" Hebrew? Loanword, naturalized, your TTS engine will pronounce it fine in English. So the classifier has to learn a boundary that's fuzzy.



Corn

Which is where I'd push back on the classifier a little. For a podcast pipeline where Daniel controls the input, the frequency-threshold approach we landed on before is probably more robust. Track which words show up repeatedly in contexts where Hebrew is expected, build a small dictionary, and let the classifier handle the long tail.



Herman

That's the pragmatic version. And it's worth noting the counterpoint from VietNormalizer. That's a zero-dependency Python library for Vietnamese TTS text normalization, and it's rule-based. Regex, CSV dictionaries, Unicode normalization. No neural anything. And it explicitly positions itself against tools that require heavy neural dependencies while covering a narrow subset of cases.



Corn

So for some normalization jobs, deterministic rules beat a chain of classifiers.



Herman

For narrow, well-understood jobs, absolutely. Vietnamese text normalization is a solved problem in the sense that the rules are known. Hebrew detection inside English text is less solved, because the boundary is fuzzy. But the lesson stands: not every problem needs a neural network.



Corn

Okay, so detection. Now conversion. Once you've found the word, you convert it to Hebrew script. That's transliteration.



Herman

And this is where it gets hard, and I want to flag something Daniel may not have fully priced in. Unvocalized Hebrew is ambiguous. Hebrew script without vowel markings, without diacritics, can be read multiple ways. There's a paper on this, LOTHM, enhancing TTS stability in Hebrew using discrete semantic units. Their finding is that Hebrew's non-diacriticized script causes inherent instability in TTS systems.



Corn

Inherent. Not a bug you fix.



Herman

The script itself is underspecified. So your recipe may need a diacritization step, or a phonemization step, not just script conversion. You're not just swapping alphabets. You're resolving ambiguity that the writing system left open.



Corn

Which means the "convert to Hebrew characters" step is really two steps, and the second one is the hard one.



Herman

And it's the one that determines whether the output sounds right. You can get the characters perfect and still have the TTS engine guess the vowels wrong.



Corn

Then the tagging step. SSML.



Herman

Right. The output needs to be formatted for the engine. SSML lang tags or voice spans. SFMS-ALR generates a unified SSML representation with appropriate lang or voice spans, synthesized in a single request. That's the standard pattern for code-switch TTS. And it's what Daniel means by "the necessary tags for Chatterbox conformity."



Corn

Except.



Herman

Except Chatterbox is English-only. That's in its README. Currently only English. Multiple people have noted that non-English input comes out accented. So this is a real constraint for the Hebrew spans. You may need a different engine for those, or you accept the accent.



Corn

That's not a recipe problem. That's a model problem.



Herman

It's a model problem, and it deserves to be said plainly. You can build the most elegant pipeline in the world and the Hebrew words will still come out sounding like an American reading a menu.



Corn

So the recipe might need to route. English spans to Chatterbox, Hebrew spans to something else, then stitch the audio.



Herman

Or accept that the Hebrew is accented and move on. For a podcast that uses occasional Hebrew words, that might be fine. Daniel lives in Jerusalem. His listeners probably know what Shabbat sounds like.



Corn

Now the big architectural question. Chaining small models versus one large model.



Herman

There's a paper for the single-model side. CS-LLM, out of ASRU. It enhances code-switched TTS in LLMs using only monolingual corpora, via a split-and-concatenate data construction strategy. So one model, no code-switched training data needed.



Corn

And the chaining side has SFMS-ALR, which explicitly argues for modularity. Engine-agnostic, no retraining, flexibility, interpretability, immediate deployability. They call it a modular baseline by design.



Herman

The honest trade-off. Chaining gives you interpretability, replaceability, debuggability. You can see which stage failed. If the Hebrew comes out wrong, you know whether it was detection, conversion, or tagging. A single large model is a black box. It either works or it doesn't, and when it doesn't you have no idea why.



Corn

For a podcast pipeline, where Daniel controls the input format and the volume is low, chaining is defensible.



Herman

For a general-purpose product, the single-model approach is winning. Fewer moving parts, better end-to-end naturalness. And there's a historical note here. Old ASR systems were always chained with a language model, usually n-grams. Then the pendulum swung to end-to-end. And now for specialized tasks it may be swinging back.



Corn

The pendulum always swings back. That's what pendulums do.



Herman

There's also a pattern worth stealing from a recent paper. Complex-text robustness, from September, evaluates multilingual TTS failure modes on code-switched expressions and introduces something called a Text Risk Score. It estimates synthesis risk from interpretable text features, no manual annotation, no model training.



Corn

A pre-flight check.



Herman

A cheap pre-flight check that flags risky inputs before you generate. That's a component that belongs in the recipe. Before you spend compute on synthesis, ask whether this string is likely to break.



Corn

I like that. It's a smoke detector.



Herman

And it's the kind of thing that only exists because someone documented their pipeline well enough to notice the pattern.



Corn

So that's the technical challenge. But the bigger question is how you package this so you can use it again, or share it with someone else. And Daniel's instinct about containerization is spot on. Let's talk about why, and what the alternatives are.



Herman

Containerization is a strong fit, and the reason is dependency hell. That's the number one barrier to reuse. Not the model. The install.



Corn

Give me the Chatterbox example, because it's perfect.



Herman

The Hugging Face thread on Chatterbox is full of people who couldn't get it running. One commenter: doesn't work with Python 3.13, numpy 1.26.4 doesn't exist, the pip install version has a bug in CPU-only mode, requires protobuf-compiler, got a weird CMake error, after an hour it still doesn't run.



Corn

An hour.



Herman

And then: "I know anger isn't productive but this is my experience almost any time I'm running somebody else's Python project. Hit an issue, back up, hit another issue, back up, after an hour it still doesn't run."



Corn

That's the whole problem in one paragraph.



Herman

Someone else in the thread: "We'll know AGI has arrived when it can figure out Python dependency conflicts."



Corn

That's the best line I've heard all week.



Herman

And the fix is documented in the same thread. We developed and tested Chatterbox on Python 3.11. That's it. Pin the version. A container that pins Python 3.11, torch 2.6, and the exact dependency set solves the number one complaint.



Corn

Which is why travisvn's chatterbox-tts-api exists. It's a Dockerized API wrapper that makes installation easier. And it's designed as a drop-in replacement for OpenAI's TTS API in local frontends.



Herman

That's the pattern. Bundle the model, the code, and a stable interface so it drops into other pipelines. And there are others. Baseten has a packaged serving implementation. Lightning AI has a LitServe-based production TTS API with Chatterbox. This is becoming the standard shape.



Corn

So Daniel's instinct toward download-on-first-run for the weights. Is that right?



Herman

It's the more common production pattern. Two options. Bake the weights into the image, which is large and slow to pull but zero-config. Or download and cache on first run, which is a smaller image but needs network access and a cache volume. Download-on-first-run is what most people do.



Corn

And the resource requirements are modest, right?



Herman

Chatterbox needs about five to six gigabytes of VRAM peak. Someone ran it on an old RTX 2060. So this isn't a datacenter problem. It's a laptop problem.



Corn

Now Replicate. Daniel says we previously told him it doesn't support this kind of chaining, and he's asking why.



Herman

The architectural reason. Replicate's Cog model is designed around a single model, single container, single prediction endpoint. Each Cog exposes one predict interface. Chaining multiple models means either calling multiple Replicate endpoints in sequence from your own orchestration code, or building a custom Cog that internally loads and runs multiple models.



Corn

And the second one defeats the purpose.



Herman

It bloats the container and you've just rebuilt the problem you were trying to avoid. So the practical takeaway: Replicate is a model hosting service, not a pipeline orchestration service. Your orchestration logic lives outside it. That's a genuine gap in the ecosystem.



Corn

And it's why people reach for containers instead.



Herman

It's exactly why. You need somewhere for the glue to live.



Corn

ComfyUI. Daniel says he feels it's an awkward fit for non-image workloads. Is he right?



Herman

He's right, and the evidence supports it. ComfyUI is fundamentally a node-graph tool for image and video generation. Someone from Rubbrband described it well: it's primarily great because it's a node-based tool, so you can chain together models, upscalers, prompting nodes. Note the examples. All image concepts.



Corn

The community is image-centric.



Herman

Entirely. SDXL Turbo, LivePortrait, video models. That's the ecosystem. Now, there is movement toward general use. Someone built a self-hosted gateway to access LLMs, Ollama, ComfyUI and FFmpeg servers, treating ComfyUI as one backend among many.



Corn

And there's Comfy Nodekit, from October last year.



Herman

It lets you build and serialize ComfyUI workflows in Python, with typed node factories. And the motivation is telling. Hand-crafting large ComfyUI graphs or maintaining JSON workflows by hand gets messy once you reach a few dozen nodes.



Corn

So they built a tool to avoid writing the JSON.



Herman

They built a Python interface to avoid writing the JSON. Which tells you the JSON is the problem. And here's the insight. ComfyUI's graph format is technically a fine way to express a pipeline. But its UX, its node library, its community, are all optimized for visual generation. Using it for ASR or TTS normalization means fighting the grain. You'd be hand-writing JSON graphs, hence tools like Comfy Nodekit, for a task that plain Python expresses more naturally.



Corn

So the recommendation is Python as the glue.



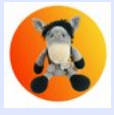
Herman

Python as the glue. Daniel's original instinct was right.



Corn

That brings us to the distribution problem. You've built this great recipe. Now where do you put it?



Herman

Let's synthesize. Format first. Four pieces.



Corn

Go.



Herman

One. A container image, Docker or OCI, as the runtime artifact. That solves dependency hell. Pin Python 3.11, torch 2.6, exact deps. Download weights on first run to a cache volume.



Corn

Two.



Herman

A thin, stable API surface. Follow the chatterbox-tts-api pattern. Expose an OpenAI-compatible or simple HTTP endpoint so the recipe drops into any pipeline. That's what made that wrapper successful. It's not that the model is better. It's that the interface is predictable.



Corn

Three.



Herman

A declarative recipe file. YAML or JSON. Describing the chain. Model IDs, versions, order, parameters, replaceable variables. This is the documentation artifact. The thing that answers "which models did I use."



Corn

And four.



Herman

Python as the glue. Keep the orchestration in plain Python. More debuggable than a node graph for this class of task.



Corn

Now the channel. Ranked.



Herman

One, a GitHub repo with the container, the recipe file, and a README. That's the default. It's where all the examples live. Two, a container registry, GHCR or Docker Hub, for the image itself. Three, Hugging Face for any model weights you fine-tune, or for the recipe card. HF is increasingly used for non-model artifacts. Four, PyPI if the glue code is a reusable library.



Corn

Like VietNormalizer.



Herman

Like VietNormalizer. Zero-dependency, standalone installable, on PyPI. That's the model for a narrow library.



Corn

And the for-yourself versus for-others distinction.



Herman

For personal reuse, a container plus a recipe file in a private repo is sufficient. You're the only consumer. For others, the documentation and the stable interface matter more than the code. The Chatterbox thread shows people will tolerate a mediocre model but not a broken install.



Corn

That's the whole thing. A mediocre model with a clean install gets used. A brilliant model that takes an hour to set up gets abandoned.



Herman

Every time.



Corn

So what's the right level of abstraction here? Because a recipe can be anything from a README to a full container.



Herman

It can. And I don't think the ecosystem has settled. It might mean a paragraph in a blog post or it might mean a signed container image with a declarative manifest.



Corn

Is it going to standardize?



Herman

I'm not sure. There's pressure toward it. Hugging Face model cards are a de facto standard for models. Something similar could emerge for recipes. But the incentive to standardize is weaker, because recipes are more personal. They encode your specific choices.



Corn

Your folder called tts_stuff_final_v3.



Herman

The balance you have to strike is between reproducibility and flexibility. A recipe that's too rigid can't be adapted. A recipe that's too loose isn't reproducible.



Corn

Which is the same problem as documentation generally. Too much and nobody reads it. Too little and it's useless.



Hilbert

You keep saying recipe. It's a script.



Corn

...A script.



Hilbert

I did localization engineering for a software company for two years. English product, twelve languages. My job was to make the translated strings fit in the boxes. You want to know what my recipe was? A spreadsheet and a regex. Column A, the English string. Column B, the translation. Column C, who broke it. And a script that ran at build time and swapped them. That was it. It wasn't elegant. But when something broke I could fix it in five minutes.



Corn

The pipeline Daniel's describing?



Hilbert

It's a pipeline to handle a few Hebrew words. I'm not saying it won't work. I'm saying you're building a machine to do what a lookup table does. The best recipe is the one that can't break.



Herman

There's a version of that argument I'd agree with. The VietNormalizer point.



Hilbert

I don't know what that is. But I spent a week once building a modular localization system. Proper architecture. Abstraction layers. And a developer hard-coded an English string into a button and bypassed the whole thing. Broke the build for three languages. One string.



Corn

The abstraction was the vulnerability.



Hilbert

The abstraction was the vulnerability. The spreadsheet wouldn't have cared. Somebody would've added a row.



Herman

Do you still have the spreadsheet?



Hilbert

It's on a drive in the other room. It's got every English idiom that doesn't translate. "Biting the bullet." "Under the weather." The German team just wrote "problem" for most of them.



Corn

That's a historical artifact.



Hilbert

It's a list of things that don't work. Anyway. I've got a sofa that needs moving before it gets dark.



Corn

Before it gets dark.



Hilbert

The light's better in the morning and I'm not doing it in the morning.



Corn

Where does that leave us. The recipe concept is real, but Hilbert's got a point buried in there. The value isn't in the architecture. It's in the documentation of what actually worked.



Herman

Which is why the recipe file matters more than the container. The container solves the install. The recipe file solves the memory problem. And the memory problem is the one Daniel actually named. He wants to remember which models he used.



Corn

One thing I'd watch. As pipelines get more complex, the ability to package and share them becomes a skill in itself. The tools and formats we pick now shape how collaborative AI development works. If everyone invents their own recipe format, we get a mess. If something standardizes, we get leverage.



Herman

The standardization pressure is coming from the pain. People are tired of broken installs. That's what'll drive it.



Corn

Thanks to Hilbert Flumingtop for producing. This has been My Weird Prompts. Check the show notes for links to the papers and tools we mentioned, and if you've built a recipe of your own, send it to us. Email us at show at my weird prompts dot com.



Herman

We'll be back soon.