Vibe Coding Challenge: The Art and The Skill

Kabui, Charles

2025-12-31

Table of Contents

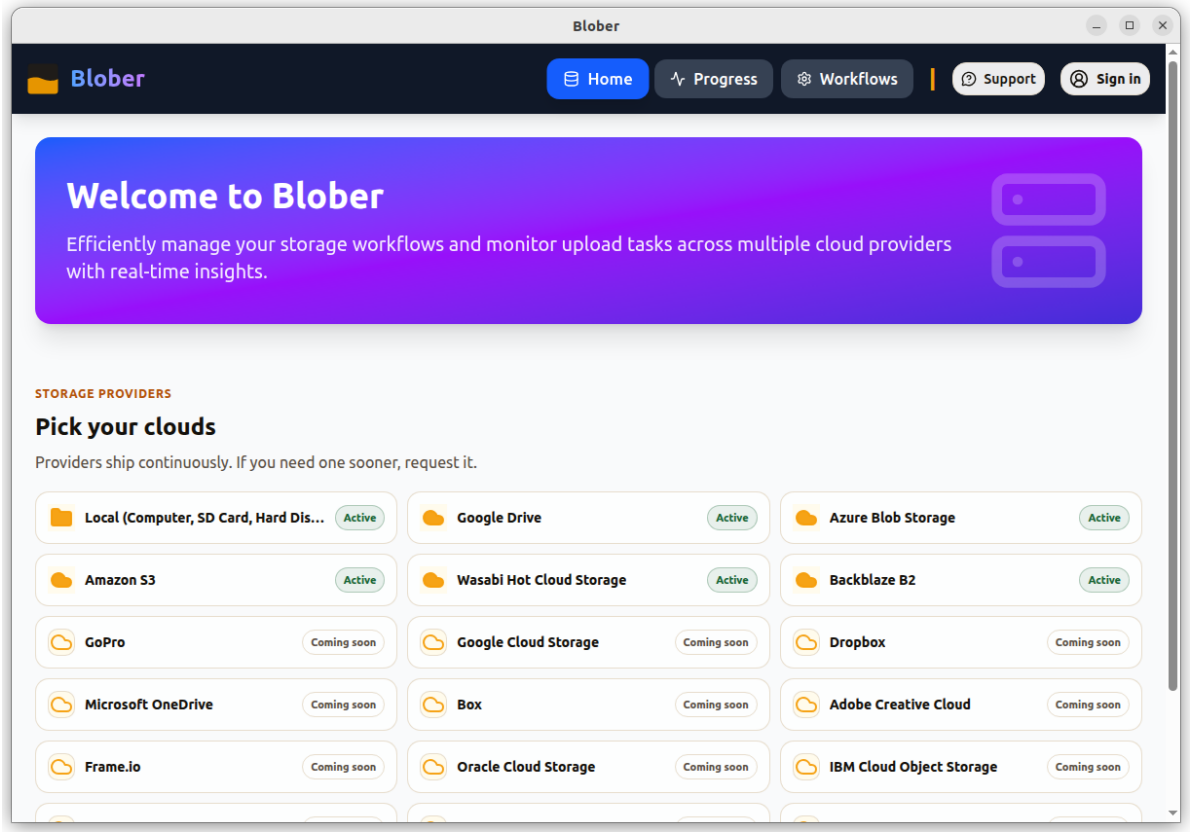

Figure 1: Blober.io App Screenshot

Archiving 1TB/Month: Why I Built Blober.io and What I Learned About "Vibe Coding"

My General Take on AI-Assisted

Generally, I have found the following to be true about AI-assisted solutions in all domains:

1. AI is good for:
    - Drafting (kick starting a new project, templating, Generating a new document, etc)
    - Proofreading (writing tests, debugging errors, etc)
    - Manipulation (summarizing text, rephrasing a document, remixing the tone or theme of a text block, improving existing code, etc)
2. AI saves you money in the short run (you dont need a website designer), but

> compensates by wasting time (you spend all the time prompting and tuning a design)
>
> Simply speaking, AI is best used as a multiplier of existing skills, not a replacement for them.

As engineers, we often pride ourselves on the "elegance" of our scripts. For months, I managed a 1TB/month pipeline of media assets (archiving them to Azure Blob Storage) using a custom Python script. It worked fairly well but had its limitations.

Click here to view the script

To run the script, the following dependencies were required:

```
python3 -m venv venv # Create an enviromnemt
source venv/bin/activate
python -m pip install --upgrade pip
pip install azure-storage-blob boto3 paramiko wakepy
```

```
{
  "connection_string": "DefaultEndpointsProtocol=https;AccountName=mystorageaccount;AccountK
  "file_search_patterns": [
    [
      "/home/user/Pictures/Vacation/*.{jpg,png,gif}",
      "/home/user/Videos/Family/**/*.{mp4,avi,mov}"
    ]
  ],
  "upload_directory_expression": "backup/{file_created}/{file_created_datetime}:{filename}",
  "storage_tier": "archive",
  "status_file": "./upload_status/status.json",
  "log_file": "./upload_logs/upload.log",
  "console_log_level": 2,
  "file_log_level": 6,
}
```

Then, you'd run the script: `python upload_script.py config_sample.json --dry-run` to preview the changes without uploading, and then when you are ready, you would do: `python upload_script.py config_sample.json`. This would upload the files matching the patterns to Azure Blob Storage, organizing them by their creation date, ending up with a structure like this:

```
backup/
   2025-12-24/
```

```
    2025-12-24_08-24-02:DSC08007.jpg
    2025-12-24_09-15-45:IMG_1234.mp4
2025-12-25/
    2025-12-25_10-05-30:HolidayPic.png
    2025-12-25_11-20-10:FamilyVideo.avi
```

Now suppose you want to run this script every night at 2 AM? Thats another setup! If you want Upload to AWS S3 instead of Azure Blob? More changes! Over time, this script becomes very complex and hard to maintain. What if you need to run the same setup on Windows or MacOS? More setup work! And at the end of the day, python isn't particularly well suited for async IO operations, so the performance is not optimal.

When you are dealing with high-volume archival, the script eventually hits a wall of **operational debt**. I realized I didn't just need a transfer tool; I needed a **state-aware workflow engine**.

I spent two weeks over the Christmas break building Blober.io. Here is the technical breakdown of the project and my honest take on the "Vibe Coding" approach that made a 14-day delivery possible.

The Problem

Cloud providers (especially Azure Blob) are phenomenal for scale but terrible for discovery. If you dump 10,000 files into a flat container, you have essentially created a data graveyard.

I needed my files organized *deterministically* as they were uploaded. My preferred pattern: `container/{file_created_date}/{file_created_datetime}:{filename}`

While `rclone` is the industry standard, achieving this level of metadata-aware path structure would require complex wrapper scripts. In Blober, this is a first-class primitive!

The Architecture: Local-First Workflow Model

Blober isn't just a CLI wrapper. It uses a persistent **Workflow/Task model** backed by a local SQLite DB:

- **Immutable Jobs:** When a workflow triggers, Blober snapshots the configuration into a "Job."
- **The Executor:** A background polling loop in Electron that picks up tasks and, crucially, manages power states to keep the machine awake until the transfer is 100% complete.
- **Metadata Engine:** We inspect file attributes via `stat()` or provider APIs to inject variables like `{file_created_date}` or `{file_size_mb}` into the destination path in real-time.

Lessons from "Vibe Coding"

Building a full-stack desktop app in two weeks requires a different approach to development. I leaned heavily into AI-assisted "Vibe Coding," and the results were enlightening.

1. The Tooling Hierarchy

Not all models are created equal for engineering:

- **The Modifier: Claude 3.5 Sonnet** remains the undisputed king for modifying existing, complex codebases without breaking state.
- **The Designer: Gemini 3** is surprisingly creative with user layouts and User interfaces.
- **The Baseline:** Python and ReactJS are now so well-supported across all models that boilerplate is essentially a solved problem.

2. The "Deep Thinking" Trap

One of my biggest takeaways: **Deep thinking/research models are often useless for code generation.** When you ask a model to "think" too hard about a coding problem, it often results into an over-engineered, verbose logic that ignores the immediate context. Quick-thinking models (like Sonnet) are superior because they iterate at the speed of the developer's intent.

3. The Prompt Engineering Hack

Without a strict framework, all models eventually drift into slopy code. My most successful prompt strategy is forcing the AI to plan before generating the code, by ending every prompt with:

> *"Before you start implementation and code generation, create a detailed multi-phase, multi-stage, and multi-step implementation plan. Before proceeding to the next phase, ask for my explicit consent."*

This "Pause-and-Evaluate" gate is the only way to prevent an AI from hallucinating a 500-line file that solves a 50-line problem.

4. The Economic Trade-off

Vibe coding is a double-edged sword:

- **The Win:** It saves significant capital, for example, you dont need to hire a UI/UX designer!
- **The Loss:** It is a massive time-sink. You trade "writing code" for "tuning prompts.You can spend hours perfecting a design.

Blober vs. rclone: An Honest Case

Blober is not intended to replace rclone. If you are running headless Linux servers with complex filter flags, stay with rclone.

Choose Blober if:

- You want a **persistent history** of every file moved (via the local DB).
- You need **human-readable guardrails** and conflict detection.
- You want **metadata-aware pathing** without writing custom bash/python wrappers.

What's Next?

The "Christmas Challenge" version of Blober is live. I'm currently looking to add support for more cloud providers and move the `sync` implementation from "planned" to "production-ready."

If you're moving high volumes of data and find yourself fighting with scripts, I'd love for you to give Blober.io a spin. I'm looking for brutal feedback on the workflow model: does it solve your organizational debt?

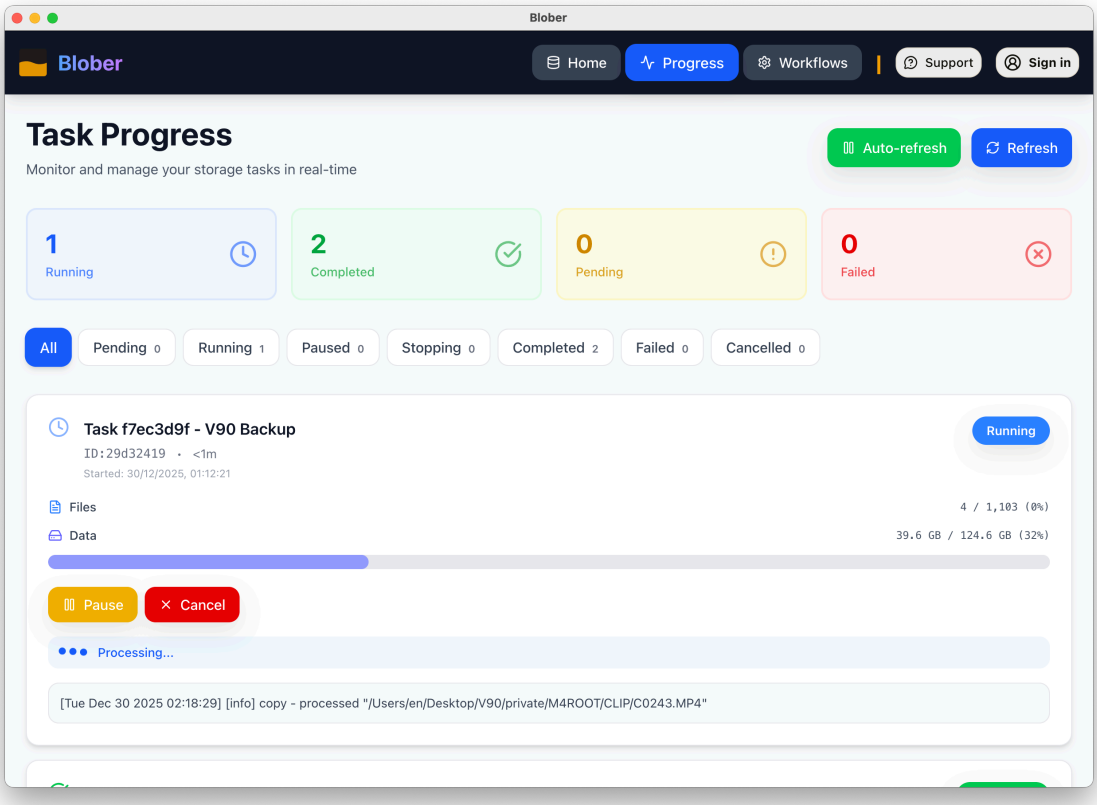

Figure 2: Progress Page

Disclaimer: *For information only. Accuracy or completeness not guaranteed. Illegal use prohibited. Not professional advice or solicitation.* ***Read more:*** */terms-of-service*