Overview

This document should highlight what steps I took to get each Docker Image deployed onto my local machine/server. The majority (70%) of the web apps were able to deploy using 'docker compose', but some others needed modifications to get them working. Following the order on the Google Sheets, I made a step-by-step instruction guide for all the successful deployed images.

P.S. The majority of these images were tested and functioned well under a Git Bash script I made that essentially mass tested each image to 1) reduce time and resource consumption and 2) verify that their containers were operating successfully. Hence, whenever in this documentation you see that it says: "Successfully ran under the original testing Git Bash script file, you have a better comprehension by what I mean." Because there are other images that will reference git bash rather than Windows Powershell, which I have used as the standard throughout my testing since I own a Windows machine.
Initially, I thought it would be nicer looking to match the Google Sheets with this document, but for more clarity, I realized it would be better to sort based on the way I tested each Dockerized web app. Therefore, I will start with the ones that successfully ran under the original Git Bash script. I will fix the reordering as time progresses.

List of my Dockerized Web Apps:

Docker Image Name
metabase/metabase
archivebox/archivebox
serversideup/all-in-one
goniszewski/grimoire
yourselfhosted/slash
hascheksolutions/opentrashmail
listmonk/listmonk
strapi/strapi
wordpress
baserow/baserow
bytebase/bytebase
freshrss/freshrss
mintplexlabs/anythingllm
ollama/ollama
farmos/farmos
odoo

typesense/typesense
getmeili/meilisearch
openproject/openproject
leantime/leantime
statping/statping
twinproduction/gatus
mediacms/mediacms
b3log/siyuan
blinkospace/blinko
linuxserver/hedgedoc
neosmemo/memos
pretix/standalone
activepieces/activepieces
apache/airflow
healthchecks/healthchecks
lycheeorg/lychee
photoprism/photoprism
bpatrik/pigallery2
paperlessngx/paperless-ngx
stirlingtools/stirling-pdf
passbolt/passbolt
linuxserver/bookstack
snipe/snipe-it
loomio/loomio
huginn/huginn
bitnami/discourse
mayanedms/mayanedms
kopia/kopia
rmcrackan/libation
machines/filestash
rocketchat/rocket.chat
reactioncommerce/reaction
kromit/titra
wekanteam/wekan

The Original Git Bash Testing Script

The purpose of this script was to mass test a series of Dockerized web applications with minimal manual effort. By automating this task, I aim to do the following:

- Identify which images could run successfully out of the box.
- Detect missing configurations such as *docker-compose.yml*
- Save time by avoiding manual pull/run steps
- Quickly surface images that required manual adjustments

Below is the script:

```
#!/bin/bash

# ------------------------------------------
# This script pulls and tests a list of Docker images.
# It attempts to run each image briefly and then stops it.
# ------------------------------------------

# Define the list of Docker images to test
images=(
 “<IMAGE_NAME>”
)

# Loop through each image in the list
for image in "${images[@]}"; do
  echo "Pulling: $image"       # Inform which image is being pulled
  docker pull "$image"        # Pull the image from Docker Hub

  echo "Running: $image"      # Inform that the container is starting
  docker run --rm "$image" &  # Run the container in the background and remove after exit
  pid=$!                      # Capture the process ID of the container

  sleep 30                    # Let the container run briefly for 30 seconds

  # Kill the container if it's still running
  if ps -p $pid > /dev/null; then
    kill $pid                 # Stop the container
  fi

  echo "Testing done for: $image"
  echo ""                     # Add spacing between each test block
done

echo "ALL DONE!"             # Final message
```

Workflow/Reasoning/Limitations Behind the Original Testing Bash Script

Here were the steps I took to use this script and why they mattered:

1. nano test_images.sh

This opens the nano text editor, creates an .sh file and pastes the code above with the image(s) wanted for testing.

2. chmod +x test_images.sh

This changes the permissions of the script, allowing it to become executed as a standalone program.

3. ./test_images.sh

This executes the script. It pulls the image(s) from Docker Hub, attempts to run it for 30 seconds to see if it initializes correctly, and stops the container to limit lingering background processes.

Reasoning- There were numerous reasons why I used this approach

- Batch Automation: Instead of testing 15+ images individually, the script automated the whole batch
- Error Surfacing: If an image was broken or missing a crucial *.env* or *docker-compose.yml,* I would immediately know by inspecting the logs.
- Minimal Resources: Containers ran for 30 seconds and auto-removed to reduce lag
- Early Filtering: Expedited the process of identifying which image(s) works out of the box or required manual fix

Limitations- Not all images ran successfully using this method, especially ones that:

- Require a *docker-compose.yml* file (services with multiple containers)
- Depend on *.env* or API keys to initialize (typesense/typesense or anythingllm images)
- Use build steps not encapsulated in base image

For these cases, I followed up by manually cloning the GitHub repo, copying *.env.example* to *.env*, or running *docker-compose up -d* in the correct directory.

Why I Used Git Bash Instead of Windows Powershell

While I used Windows Powershell for everything else, I chose Git Bash for running the test script because it offered key advantages. They include:

- Unix-style scripting compatibility: The `test_images.sh` script uses standard Bash syntax (like `for` loops and `$!` for background processes), which is native to Git Bash but not to PowerShell.
- Cross-platform portability: Bash scripts are widely used across Linux, macOS, and CI/CD pipelines. Writing the script in Bash made it reusable outside Windows if needed.
- Simpler Docker interaction: Git Bash provides better compatibility with Docker CLI usage for scripting purposes compared to PowerShell, where output formatting and process management can be inconsistent.
- Reduced complexity: Instead of adapting bash logic into PowerShell's syntax-heavy structure, using Git Bash let me stick to examples and community-supported methods found in most Docker documentation.

All of My Web APPS

Images That Ran Successfully Under ./test_images.sh (41 total)

metabase/metabase
archivebox/archivebox
goniszewski/grimoire
yourselfhosted/slash
hascheksolutions/opentrashmail
listmonk/listmonk
strapi/strapi
baserow/baserow
bytebase/bytebase
freshrss/freshrss
ollama/ollama
farmos/farmos
odoo
getmeili/meilisearch
leantime/leantime
statping/statping
twinproduction/gatus
mediacms/mediacms
linuxserver/hedgedoc
pretix/standalone
activepieces/activepieces
apache/airflow
lycheeorg/lychee
photoprism/photoprism
paperlessngx/paperless.ngx
stirlingtools/stirling-pdf
passbolt/passbolt

vaultwarden/server
lissy93/dashy
glanceapp/glance
linuxserver/bookstack
snipe/snipe-it
bitnami/discourse
mayanedms/mayanedms
kopia/kopia
machines/filestash
rocketchat/rocket.chat

Images That Ran Successfully NOT Under ./test_images.sh (9 total)

wekanteam/wekan
This image did not run under the bash script due to its multi-container architecture requiring Docker Compose. Manual setup with `docker-compose.yml` was necessary to orchestrate dependent services like MongoDB.

kromit/titra
This image did not work because the deployment required manual invocation using a specific Git Bash command documented in the README. The container failed under automated testing due to missing environment configuration and startup instructions.

reactioncommerce/commerce
The application did not initialize successfully in automated testing due to incomplete service startup defined in its `docker-compose.yml`. Manual environment variable setup and a secondary command (`pnpm run start:dev`) were required to launch the app fully.

neosmemo/memos
The container could not be deployed via script due to missing database migration steps and unconfigured environment variables. Manual `.env` updates enabled successful deployment.

blinkospace/blinko
The image failed under script-based testing due to a multi-stage build requirement and external PostgreSQL dependency. Proper deployment necessitated manual cloning and orchestration with Docker Compose.

b3log/siyuan
This image required the manual addition of a `docker-compose.yml` file to function correctly. The script-based method failed due to missing configuration.

serversideup/all-in-one
Although the container runs, its multi-port architecture and lack of default service routing prevented full validation under the testing script. Manual inspection confirmed successful deployment.

typesense/typesense
The container could not start properly due to initialization logic residing in a nested directory and not in the root directory. A manual launch resolved these issues.

healthchecks/healthchecks
The image build failed due to Windows-style line endings in the source Python files. Resolving the issue required converting line endings to Unix format before building the image manually.

Summary

Out of 50 tested Docker images, 41 deployed successfully via an automated Git Bash testing script. The remaining 9 required manual intervention due to limitations such as multi-container dependencies, missing or misconfigured environment variables, unsupported build structures, or source code formatting issues. These results highlight the necessity of robust orchestration tooling and deeper configuration analysis for effective agentic automation. I will now attempt to modify my bash script to allocate for needs as such so that I can consider how our AI agent should function.