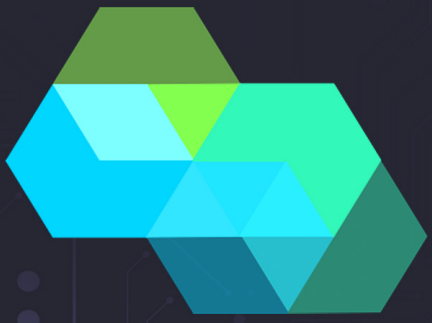
LEVEL UP WITH



EARTHLY

REPEATABLE BUILDS FOR A CONTAINERIZED WORLD

PETER MEMBREY
AND SHAUN SMITH

Level Up with Earthly

Repeatable Builds for a Containerized World

Peter Membrey and Shaun Smith

This book is for sale at http://leanpub.com/earthly

This version was published on 2022-03-31

Contents

Is this book for me?

Welcome to Level Up with Earthly! When we wrote this book we had a very particular audience in mind, both in their current Earthly expertise (complete beginner) and their preference for fully fleshed out concepts (we won't skip steps). If at any point when reading this book you feel that we're:

- being condescending
- explaining in depth what seems like obvious concepts
- using too many words
- including too much backstory

then congratulations friend, this book is definitely not written with you in mind! You'll probably have a much better experience with the excellent online documentation[1] and the many examples[2] on the Earthly website[3].

For those that enjoy a more casual writing style, stories of geeky battles against stubborn code, step by step instructions and some hand holding, then we think you'll not only be able to get a lot out of this book, but you'll probably find it at least somewhat entertaining too!

Onwards!

[1] https://docs.earthly.dev/
[2] https://docs.earthly.dev/docs/examples
[3] https://earthly.dev/

Preface

Thank you for buying Level Up with Earthly, our first book on the Leanpub[4] platform! We greatly appreciate the support and faith that you've shown in us, and we promise to do our very best to make sure that that faith has not been misplaced!

Why should I buy this book now?

Our approach is to publish new content on a regular basis with a preference for little and often. This might mean we release a half written chapter (clearly marked as such) because we believe the content is already useful and it could be just the thing you've been waiting for!

We'll also release updates and fixes (such as for typos or errors in example code) as soon as they've been identified and fixed. We are avid readers ourselves and we are all too familiar with how frustrating a poorly edited book can be. Even if you find a single typo, we definitely want to hear from you!

Although Earthly is already mature enough for production use, it is under heavy development with new features being added constantly. We keep track of these developments and update the book as those features are released. You'll get ever evolving content that's always hot on Earthly's heels.

[4] https://leanpub.com/

How can I be sure that the content is fresh and accurate?

We are honoured to be able to share that we are collaborating closely with the Earthly core team on this book. This means that we will be aware of new features and can publish new content on them shortly after release. We also actively seek their guidance and advice which helps capture their deep knowledge of Earthly in these pages for you to enjoy.

The Earthly core team has also offered to provide the technical review for this book. That means that before this book can be finalised, it must receive their seal of approval for quality and correctness.

We are extremely excited to be working with the core team and we already know that their contributions are going to be super valuable!

How can I be sure you'll finish the book?

There's always a risk with these sorts of projects, but both of us are experienced writers with a passion for tech and sharing what we've learned. We have everything we need to ensure our success.

However, despite our best efforts, things might not go according to plan. If this is a concern for you, you might want to wait until the final book is published before buying.

What sort of feedback do you value?

All feedback! Seriously, the more feedback the better! We want to know where we're doing a great job, what you'd like to see more of and conversely what you would like to see less of. Your feedback doesn't have to be an essay or a detailed bug report. A simple note of "There's a typo on line 10" is immensely valuable.

We use Earthly almost every day but because we toil away in our own little corner of the universe, it is inevitable that we're going to miss things. Some of the things we miss will be cringe worthy. So, if you've spotted a gap or have any anecdotes, awesome ideas or even an example that you think just has to be covered, then please please please don't keep that to yourself! Get in touch!

As a small token of our appreciation (and only with your permission of course) we will include your name in an Appendix of contributors. For cool quotes, example files or anecdotes we'll also cite you as the source (again, with your permission).

Why are you writing this book?

We're writing this book because when we were introduced to Earthly, we wanted to buy a book and learn more about it. Alas no one had written one yet. Earthly is so new and fast paced that it is not yet a good fit for traditional publishers (we know, we asked them). Two of those publishers that we have a great deal of respect for both independently recommended Leanpub, and it seemed to us like a perfect match. We could move fast, but also keep the quality bar high.

Does it replace the online documentation?

Definitely not. Earthly's documentation is exceptional and the quality and depth of the examples is first rate. It's mostly how we learned Earthly ourselves and this book will be very much influenced by its content.

But as good as that documentation is, it's more of a reference, the place you go when you have a pretty good idea of what you're looking for. This book on the other hand is the grandfather sitting in a high backed chair next to a crackling fire in the hearth whilst he reads about the myths and legends of Earthly to those who have gathered round. Okay, that's a rather flowery way of saying that this book will help you get the most of the online documentation.

The Landscape

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Introduction

Earthly is going to change how you think about building software. At least, that's the way it was for me. It didn't happen all at once, and I must admit, the first few times I checked out the website, I really didn't get it either. I mean, it sounded like a cool idea, but I couldn't see how it was different from what I was already doing with my build system. Wasn't it just Docker with a bit of window dressing? At the time it seemed very much like a solution looking for a problem. That is of course, until I actually tried it…

The first victim of my Earthly campaign was a cool little system tool I was working on. The problem was that it was now ready to share with others, and although I was using Manjaro Linux, I knew they were using Ubuntu. Previously I had whipped up a Dockerfile for this, and had written down the incantation to copy the needed files out of the container once done. It definitely felt hacky, like I was trying to make it do something it wasn't really designed for.

At that point I decided to give Earthly a go, and I'm so glad I did. Unlike Docker which is all about creating repeatable running systems, Earthly is entirely focused on creating repeatable builds of software. This means that you don't talk in terms of containers or images, you talk in terms of build artifacts and build targets. It may seem like it's just word play, but it has a significant effect on how you'll design and structure your builds. What makes sense for an Earthly build might be quite different from how you would have done it with pure Docker.

The rest of this chapter will get you oriented a little more with Earthly and will tackle the Docker question head on. We'll look at previous generation build systems, what many people use today and the sort of flexibility that Earthly will bring tomorrow (or later on this afternoon if you're feeling adventurous).

Can't I just do it with Docker?

Okay, let's take this one head on. Over the last year or so I have had the opportunity to talk about Earthly with a lot of people. Without fail, I always get one of two reactions. Either that say "Wow! That's amazing!" or they say "But wait, can't I just do that with Docker and some scripts"?

I think one reason that people jump to this conclusion is that they are very familiar with Docker and once they hear containers mentioned, they tend to switch gears and start thinking through how they would do software builds in Docker. And that's a fair position to take, especially with modern CI tools taking great advantage of Docker for exactly this purpose. Surely all that is needed is a little bit of bash scripting, and you're done. Why would you bother with Earthly?

As we touched on before, as powerful as Docker is, it's not designed for creating repeatable builds. With enough work you can leverage Docker to create such a build system (say hello Earthly), but there's a lot more to it than just copying out a software package. If you want to manage dependencies, build targets that can span across git repositories and get safe but effective caching, there's a lot more to be done. If you really want to go down this path, be warned. All you would ultimately end up doing is investing a huge amount of time and effort to create a poorer implementation of Earthly.

In short, Earthly provides everything you need out of the box and it already works extremely well. Trying to make a custom build system is a project in and of itself, so you're probably much better off using a system that already exists. Of course, if you really really want to work on a build system, Team Earthly is always happy to receive pull requests!

The bad old days

It wasn't all that long ago that software builds where rather haphazard. Many companies didn't have any formal way of building software and some (well, actually a lot) of companies didn't even have source code control. What would be unthinkable today was business as usual back then. The software industry was still only just maturing outside of a few large corporations like IBM and Sun, and there just wasn't the tooling or community that we have today. The Internet too was a very different place!

These new software companies faced a lot of challenges. It wasn't unusual for example for only a single person to know how to release a piece of software, or that the software could only be built on one machine. There was usually an ancient PC in the corner that was kept around to debug software because no one knew how it worked or how to replace it. Everyone just hoped that machine would keep on going.

It wasn't just builds that were problematic, doing testing could be equally miserable. The phrase "works on my machine" was said without a hint of irony. Just getting the software to install (does anyone remember having to install Visual C++ runtimes or the latest version of .NET? No? Just me?) could take days of back and forth, and as you were likely to be testing a snapshot of the software, you never knew for sure what exactly you were testing.

I could talk about the fun and games involved in actually performing a release, but I'll spare you the trip down memory lane. Suffice it to say, that was usually when it was discovered that half the things weren't compatible with the other half, and that at least a number of previously fixed bugs had crept back in. Those days (and nights) were full of pizza and coffee.

Today

Thankfully things today are infinitely better than they were. It's now rare to find code that isn't under source control, and most places use some sort of build system. Two big offerings in that space are CircleCI and GitHub Actions. Both offer ways of executing code (and many other tasks) as an automated part of committing or merging code. By centralising these systems and keeping the knowledge on how to build the software in code, we at least don't have to worry about how we're going to build our software.

Containers and Docker had played a huge role here. Previously either dedicated servers or VMs were required and those are fairly expensive to spin up and on demand. They're also resource intensive in that a significant amount of resources is wasted during the build (dedicated server) or overhead from the virtualisation eats into the available CPU and memory (VMs). Containers on the other hand give very strong isolation and are very fast to start and stop with very little in the way of overhead. This brought economies of scale that made modern build systems possible.

However, this modern approach is far from perfect, and there are a number of problems that remain unresolved.

Proprietary builds

The first challenge is that each build system works differently. CircleCI is very different from GitHub Actions both in how it's designed and how a build is orchestrated. That in itself might not seem too terrible as after all, you're probably going to standardise on a single platform. Unfortunately it can bite you in a two key ways.

The first and most obvious way it can bite you is that you're somewhat locked in to a given provider. Migrating a classical build

system was tedious enough, no one is likely to volunteer for the job of migrating everything again to a complete new architecture. As more and more projects use the infrastructure, it will get harder to move to another provider which means when they increase their prices, you're pretty much stuck.

The second and more subtle issue is when you want to do repeatable builds. This is where you use two completely independent build systems to create two bit for bit identical files and it's a great way to gain confidence on the security of a given build. But creating and maintaining two very different build systems and then try to get them to produce identical output files, is for the vast majority of people simply too much to ask.

Local CI

Being able to build locally is vital for quick debugging and being able to get bugfixes and new features out quickly and reliably. Here again we hit the problem of proprietary build systems because even though there is some local tooling available, it just isn't the same as the online environment. This means that most developers are building with a system that is not representative. It's not that unlikely that each developer will have their own way of building and testing too.

Ideally of course you'd want local builds to be identical to the CI builds but good luck trying to pull that off. In fact most developers that I spoke to were resigned to the fact that building locally and on CI were simply destined to be different (they mar or may not have enjoyed the ensuing conversation).

Diverse builds

These days an application can consist of many moving parts and often these require very different environments. For example,

maybe your image pipeline requires `npm` and the latest Ubuntu release, but the script that parses some legacy data only runs on old version of CentOS. And that's before we have to build our core product on Debian because that's what the servers run. This becomes unbearably frustrating very very quickly.

Now admittedly the above example is contrived, but it is a scenario many of you will be familiar with. Currently you might use VMs or even containers to help with this, and perhaps its even automated, but it still feels a bit like it's held together with string and sticky tape. It's not just the plumbing though, it's being able to monitor and gracefully handle failures or even spot them in the first place.

Project layouts

Diverse dependencies mean diverse locations. Most commonly this is a collection of repositories on GitHub which are then pulled in to the main project as dependencies. But how should you do this? You could use git sub modules, but those are both loved and hated in equal measure. You could check out the repositories as part of your build script, but that then entwines the structure of the code to how its built. You can also let the build system handle it, but then how do you also make that work locally?

Let's assume though that you've gotten your repositories in place and you're happy with whatever process you use. The next joyous challenge is to figure out how to wire up these independent projects. Even projects that simply provide libraries can have complicated build processes. WolfSSL for example makes significant changes to what is build and available in its public headers based on how it is configured with `./configure`. So you still need to go through and build these individual dependencies before you can even think about how to actually use them.

As for trying to make them all use the same compiler or the same build flags, now you're into murky scripting territory. It's doable

but unpleasant and it's easy to make a mistake. You've also now got a master build script that is tightly coupled with this project and these dependencies and that tends to make it fragile. Changing out a library or moving to a new build process would likely be a lot of work.

We didn't even get to talk about the mono repo and the similar yet different problems that can cause when trying to keep your sanity.

Parallel builds

Modern machines have lots of CPU cores and it makes perfect sense to want to use them to get the best performance out of your builds as possible. Doing this within a project is pretty straight forward. Thanks to the job server protocol, `make` can easily spread out the build steps to multiple processes. `CMake` does something very similar. Because they know how the project needs to be built they can figure out which bits can be done in parallel and which can't.

Again this breaks down somewhat when we cross project boundaries. Each dependency needs to be built before the thing it depends on - but each of those might have dependencies. How can you be sure that you're really building things optimally? If you're too aggressive you might create some really nasty and hard to find bugs in your build, but if you're not aggressive enough, you'll increase the lengths of your builds, often significantly.

There are tools to help with this such as Google's Bazel. It's designed to work with vast amounts of files and dependencies and handle them well. It's often joked that either you can have it done fast or you can have it done right. Bazel's tagline is `{Fast, Correct} - Choose two` which is awesome. What's not so awesome is that if you want Bazel's power, you pretty much need to migrate to Bazel. Does your language have great build tooling like Rust, Go or (arguably) Node? Pity, you won't be using those with Bazel.

Now Bazel is, to be fair an impressive project, but it really requires

upfront dedication to migrate to it in order to see the best results. Even if you're keen to do it, good luck getting everyone on all the other teams to migrate with you and if it's an upstream dependency? It might be game over before you've even started.

It's not that bad!

Okay, I admit it, it's possible that I'm being just a little overzealous with the doom and gloom in that last section. That said, these are all real problems I've personally battled with and I know many people who have similar tales to share over a beer or two.

So it's not all bad by any means but let's just say that it's going to get quite a lot better…

The future (aka Hello Earthly!)

Although I don't have a crystal ball that lets me tell the future (despite many years of trying to acquire one), I'm pretty confident about this section as it's very much talking about the near future. In fact, that future is very near if you're reading this because you can get them all by switching to Earthly today!

Build the same way everywhere

In the last section we talked about how much of a pain in the backside it is to set up and learn most CI systems, and that's before you decide you need two of them. And then there's the ability for your team to run the builds locally, and what if they're using different operating systems too?

Earthly has you covered! It's not only easy to create your environment (an `Earthfile` is very similar to a `Dockerfile`), but they even

have copy and paste instructions for the popular CI systems. Even more impressive is that these commands actually work!

It gets better - wherever you want to build, be it locally or remote, Linux or Mac, the command to kick off an Earthly build is exactly the same. There's a lot of understated benefits from this:

1. Once you know Earthly, you don't need to adjust your approach
2. With a simpler standard process, you get shorter and easier to follow documentation
3. With simpler documentation, you get faster and easier on-boarding and less mental overhead
4. Everyone on the team can help everyone else
5. It's trivial to add a new CI system to your pipeline
6. Developers can share build caches… but that is a story for later!

Work effortlessly with poly or mono repos

Google and -Facebook- Meta have popularised the idea of having only a single repository where all of your code lives. This allows everything to be built and managed together and cross project changes are pretty straight forward. This approach is useful too when you have lots of teams working across projects where coordination of changes (particular if you have to deploy a number of different systems in lockstep) would otherwise be a nightmare.

The more traditionally approach is to store one project in one repository. This often means that you need to involve multiple repositories even for a single build because projects naturally depend on each other. For example, you might have a C library in one project and an application that uses it in another.

Now, whichever of these approaches you choose, you need a way to wire things up so your builds work nicely together, whether that's

a single mono repo or dozens of poly repos. Whilst most build tools are designed for one or the other, Earthly allows you to easily set up your builds no matter which approach you choose. In fact, it's perfectly happy to combine the two if you'd like - it really doesn't matter to Earthly!

Use existing tooling for your language

Earthly acts like a wrapper and something of an intermediary or fixer. It lets you take the ideal build environment for your language of choice, along with any special tooling and wraps it all up for you. This means you can tailor your builds to get the best of your language without having to give up your tooling for a better build system.

Earthly can pull this off because it doesn't replace your existing build system, it wraps around it. That way no tooling needs to be replaced. In fact, Earthly isn't even that aware of what you're running inside it. It just gets out of your way.

Easily manage cross dependency parallel builds

Because each build step has a known relationship to other build steps, it is possible for Earthly to calculate which steps need to be completed in which order, and which if any can be done in parallel. This all happens fully automatically and does not depend on the language or build tool of choice.

This by itself is really cool, especially as you get all of this effectively for free. But it starts to get really interesting when you consider that the external dependencies you use could also provide you with a build target. Earthly can then reach across multiple repositories, owned by multiple organisations, resolve all of the dependencies and then start building them all at once.

Ultra caching

Caching is always hard to get right in a build system. Trying to track what has changed so that a new build can be kicked off is a lot harder than it sounds. This is especially true when you only want to rebuild the small changes and not everything in the project. Part of the reason so many different build systems exist is because each tries to make this a better and faster experience. With more complicated code bases, it really matters if your recompiles take half a second or ten seconds. This is one of the killer features of the Meson build tool, especially when combined with the Ninja build engine. After all, the fastest way to do something, is simply not to do it. If you can avoid doing a rebuild, then it will be infinitely faster than any rebuild could ever be. That sounds wonderfully philosophical but it is actually quite applicable to many real workloads, and building software is one of them.

Cue Docker and the wonderful benefits of images. These snapshots are able to capture the state of a system in a reusable and efficient way. Normally this is used to get a container into a certain state for running an application, but Earthly can take advantage of this feature too. If a build step completes, then it can simply store the image that was created at the end of that step. Now, as long as none of the inputs to that build step change (or admittedly any of the steps it depends on), we never have to build this step again.

Hang on, how does it see changes?

Spotting a file has changed is really as simple as checking the last modified time. What's tricky though is knowing what those changes mean for the build itself. There's a reason why we (or at least people of my generation) are trained to run `make clean` before a `make build` - it's because you can never be sure if `make` has got it right.

So how is Earthly different? Well, it knows what files have changed, just like `make` does. What's different is that all Earthly has to do is invalid that build step and any step that depends on it. This is guaranteed to catch all the things the file changes could affect, whilst also ensuring the previous steps are fine because those files weren't part of their inputs.

That make's Earthly's approach simpler, safer and easier to reason about. If you're used to `make clean` it's probably going to be faster too!

Getting up and running with Earthly

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Overview of Docker concepts

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Repeatability and reproducibility

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

The Basics

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Breaking it down

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Args, Env and Secrets

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Custom commands

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Advanced

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Running Docker / Docker Compose / Terraform

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Earthly accounts / orgs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Cloud Secret Storage

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Remote Caching

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Building with artifacts / Targeting (mono repo vs poly repo)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Debugging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Running commands locally (with conditionals)

Advanced config settings

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Multiplatform builds

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Appendices

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/earthly.

Contributors

From typos to amazing anecdotes, this book has greatly benefited from the help and support of the Earthly community. These pages are where we acknowledge those efforts and the people behind them!