Rules Tofu

Engineering Design Document

Jack Bradshaw

Sep 2025

Preface

This document records an attempt to build a HCL compiler and runtime, with the ultimate goal building Open Tofu programs with Bazel. It has not been implemented and the document is incomplete because deep into design an issue was found that invalidated significant portions and severely diminished the feasibility of the entire project. It now serves as a record of what was tried and why development was stopped.

At a high level the design involved building a system to download Tofu dependencies and integrate them into Bazel eternal modules, building a compiler to package Tofu sources in a deployable archive, and building a runtime to execute archives. It would have allowed users to treat Tofu as a compiled system, compose and reuse configurations, and use Bazel as usual. The design was almost complete but unraveled when the full details of modules were discovered.

Modules are documented in the background section, but abstractly, a module is a reusable unit of code. Modules may reference other modules to create a dependency graph of modules, and their references may point to the local disk or remote locations. After designing the majority of the solution, it was discovered that module imports are not constant and may use variables and constants in their expressions. Furthermore, modules may be used with loop structures and dictionaries to effectively declare lists of modules with arbitrary and complex configuration per module. An example is shown below:

This introduced significant complexity into the solution. Downloading dependencies remains feasible, but the arbitrary and variable nature of module statements eliminates the ability to easily patch them to redirect to local directories at runtime. This invalidates large portions of the intended solution, and while there are viable workarounds, they are fragile and likely to break over time.

I have come to the final conclusion that while Bazel-Tofu integration remains feasible, it comes with considerable risk, as the ecosystem itself is not well suited to hermetic development, and by design allows arbitrary code loading at runtime. I have instead decided to try Pulumi, as it effectively works as a library for various programming languages, and allows IAC without maintaining this ruleset.

What follows is the original incomplete/draft design, abandoned at the point where the issue was discovered. It contains some grammatical errors and is a little rough around the edges because there is no point spending more time on it. It may contain useful information for others approaching the same task though.

Introduction

Open Tofu (hereafter just “Tofu”) is an open source ecosystem for infrastructure as code (IAC). It offers considerable benefits to platform engineers, and is a common choice for teams who require an open source tool (i.e. not Terraform), but integration into Bazel is limited. An existing community-maintained ruleset (rules_tf) provides linting and source validation, but has minimal support for compilation (packaging files) and no support for external dependency management, meaning an E2E build is not presently possible with Bazel. Engineers must instead resort to manual build orchestration or other build tools, which introduces the possibility of human error and prohibits full Bazel monorepo development. This new ruleset was created to provide full Bazel support for Tofu, and it supports the following operations:

- Importing external Tofu dependencies into the workspace so they can be referenced as targets.
- Compiling Tofu sources into libraries so they can be exported and referenced in other targets.
- Compiling Tofu sources into binaries so they can be executed to provision infrastructure.
- Statically validating Tofu sources and running tests against Tofu sources so the contents can be verified and maintained.

Together these operations unlock the full potential of Bazel and IAC.

Background

This section provides background on Tofu for readers who are familiar with Bazel but not Tofu. It does not cover Bazel itself, and the Bazel docs are recommended prior reading. Furthermore, it covers areas of Tofu that are critical to Bazel integration and elides the rest, so the Tofu docs as recommended for broader Tofu understanding. A few custom terms are defined for concepts that are critical to Bazel integration but have no term in native Tofu.

Overview

Tofu is an interpreted language for provisioning infrastructure, and it involves four main components:

1. Tofu Providers, which are supporting utilities for interacting with specific providers.
2. Tofu Configurations, which are collections of infrastructure configuration files.
3. Tofu State, which is a snapshot of provisioned infrastructure.
4. Tofu CLI, which is a general purpose utility and interpreter for the ecosystem.

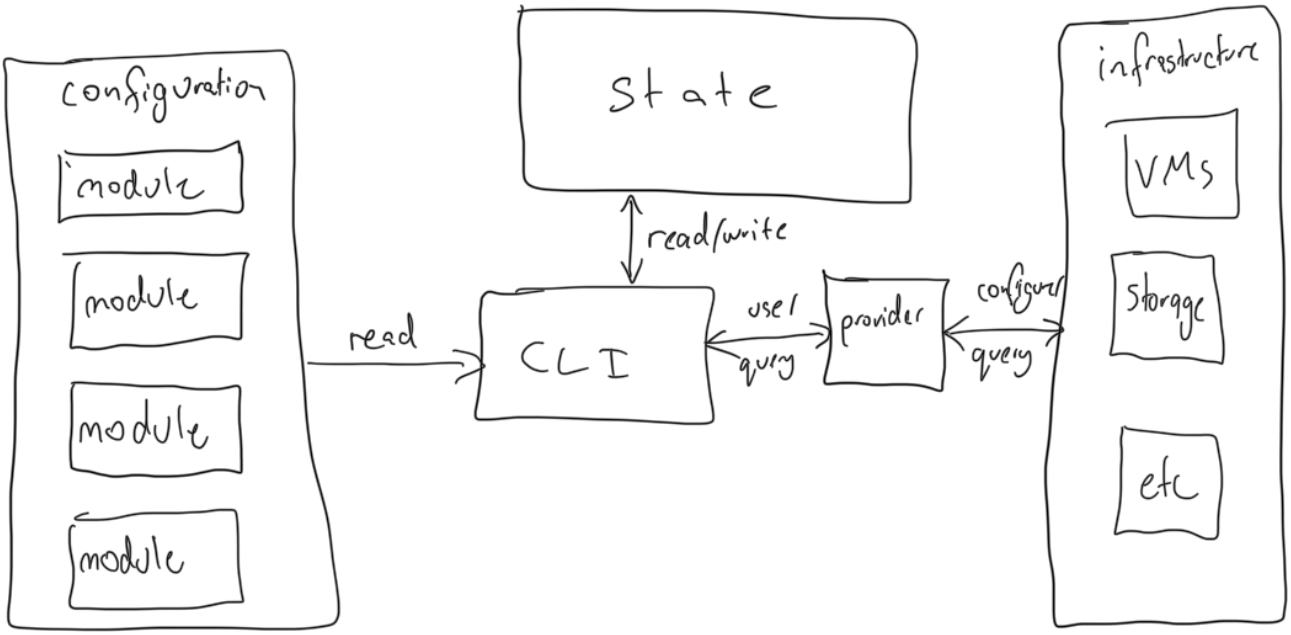
Tofu Providers are pre-built binaries that modify infrastructure based on configurations. They are they executable programs which translate the desired state to actual provisioned infrastructure. Tofu Providers can be written in various programs and communicate with the Tofu CLI via a pre-defined interface.

Tofu Configurations are collections of HCL source files and other data files which define the desired state of infrastructure. A configuration is effectively a declaration of the intended state, but not an imperative system for reaching that state. Tofu Configurations are structured into modules, which are reusable units of source code.

The Tofu CLI provides core functionality that all Tofu programs use, and is a critical component of the ruleset. It downloads external dependencies, validates and lints sources, and interprets sources to provision infrastructure. It is available from various sources including GitHub releases.

Tofu State describes infrastructure as it is currently provisioned. It's used by Tofu to track what has already been provisioned and effectively make changes when configurations change. For the most part, state is not a concern of the ruleset, and is considered a runtime implementation detail.

Together these four components make a Tofu program. Users typically select providers based on their infrastructure choices, write configuration files to declare the infrastructure they desire, execute configurations with the Tofu CLI, and use Tofu state to record what was provisioned.



Tofu Providers

Providers perform the actual CRUD operations to provision/decommission infrastructure, but they are not invoked directly by users. Instead, users write declarative configurations and let Tofu determine which calls to make. They are available for large cloud platforms (GCP, AWS, Azure), traditional hosting services (e.g. DigitalOcean), and various other services. Using providers, Tofu is able to connect the declarative world of infrastructure configuration with real infrastructure.

Each provider is effectively as a set of precompiled statically linked binaries, with one binary for each supported OS/architecture, and they are distributed via remote registries. Providers are identified by their namespace and type, and registries are identified by their hostname. For example, the hashicorp/aws has the namespace “hashicorp” and the type “aws”, and it can be downloaded from the public registry, which has the namespace “registry.opentofu.org”. These specifications are used in configuration files to declare provider dependencies (detailed below in Tofu Configurations)

When providers cannot be sourced from remote locations at runtime, filesystem mirrors can be used, which are effectively redirects from remote locations to local paths. Mirrors are declared in tofurc files and passed in to the Tofu CLI at runtime, for example, below is a `tofurc` file that redirects all providers to `/dev/disk2/providers`:

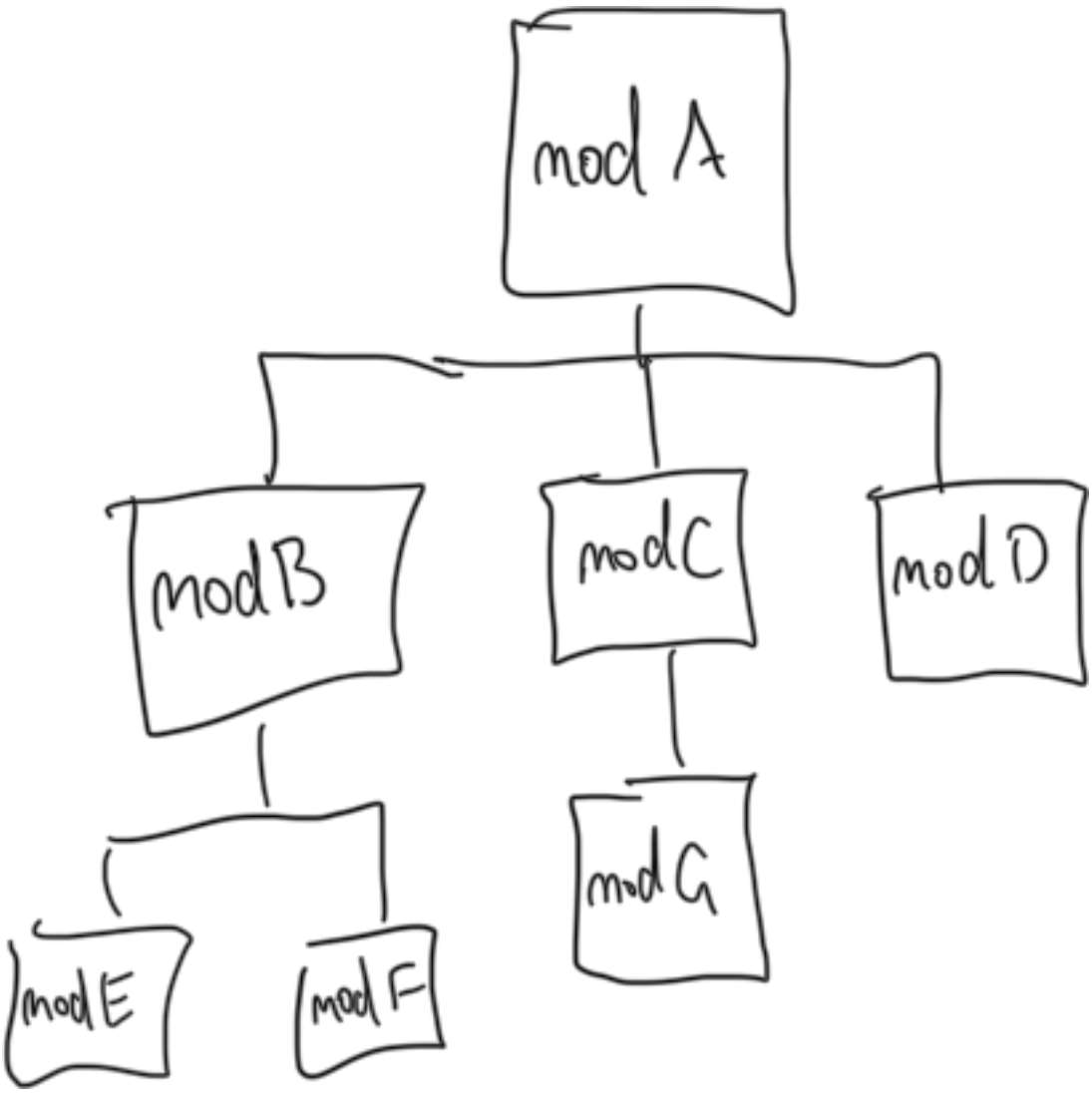
```
provider_installation {
  filesystem_mirror {
    path = "/dev/disk2/providers"
    include = ["*"]
  }
  direct {
    exclude = ["*"]
```

```
  }
}
```

Note: The filesystem mirror directory must follow a strict format defined by the Tofu CLI. Each provider binary must be stored in $mirror_dir/$hostname/$namespace/$type/$version/$os_$arch, where the mirror dir matches the redirect, and the other values derive from the binary itself. This is referred to as the unpacked format.

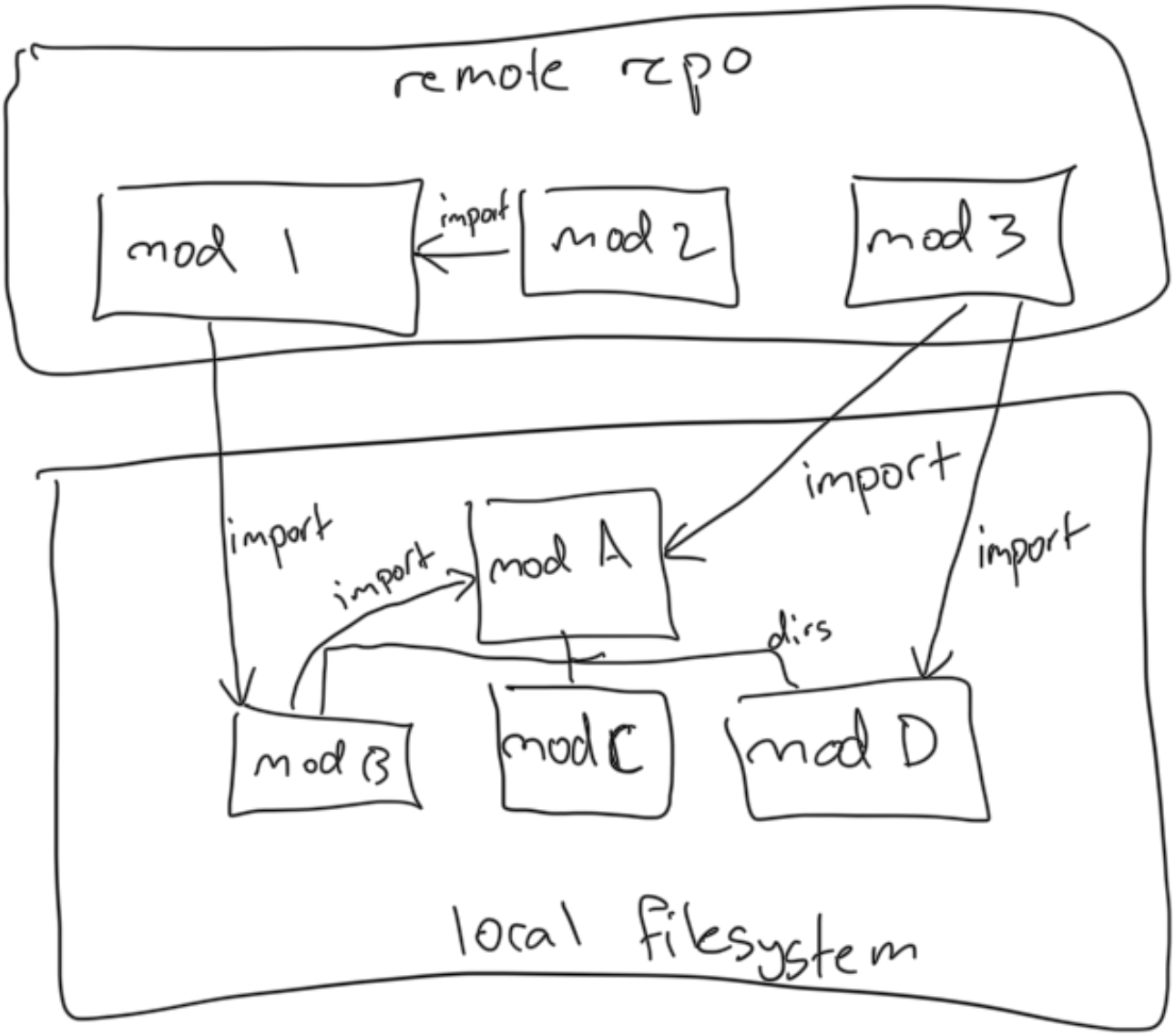
Tofu Configurations

OpenTofu configurations declare the desired state of infrastructure without specifying how that state should be reached. They are written in HCL, and since OpenTofu is interpreted, configurations are organised and distributed as directory trees of source files without an intermediate compilation step. Each directory in the source tree is a “module” in OpenTofu terminology, and when interpreted by the CLI, its contents are merged into a single logical unit. As such, a module is closer to a Java source file than to a Java package, since the division into multiple files serves readability rather than functional structure. Every Tofu plan necessarily has a root module, which is defined as the directory where the Tofu CLI is invoked, meaning their status as root modules are not details of modules themselves, but rather how the configuration is used.


mod A
mod B
mod C
mod D
mod E
mod F
mod G

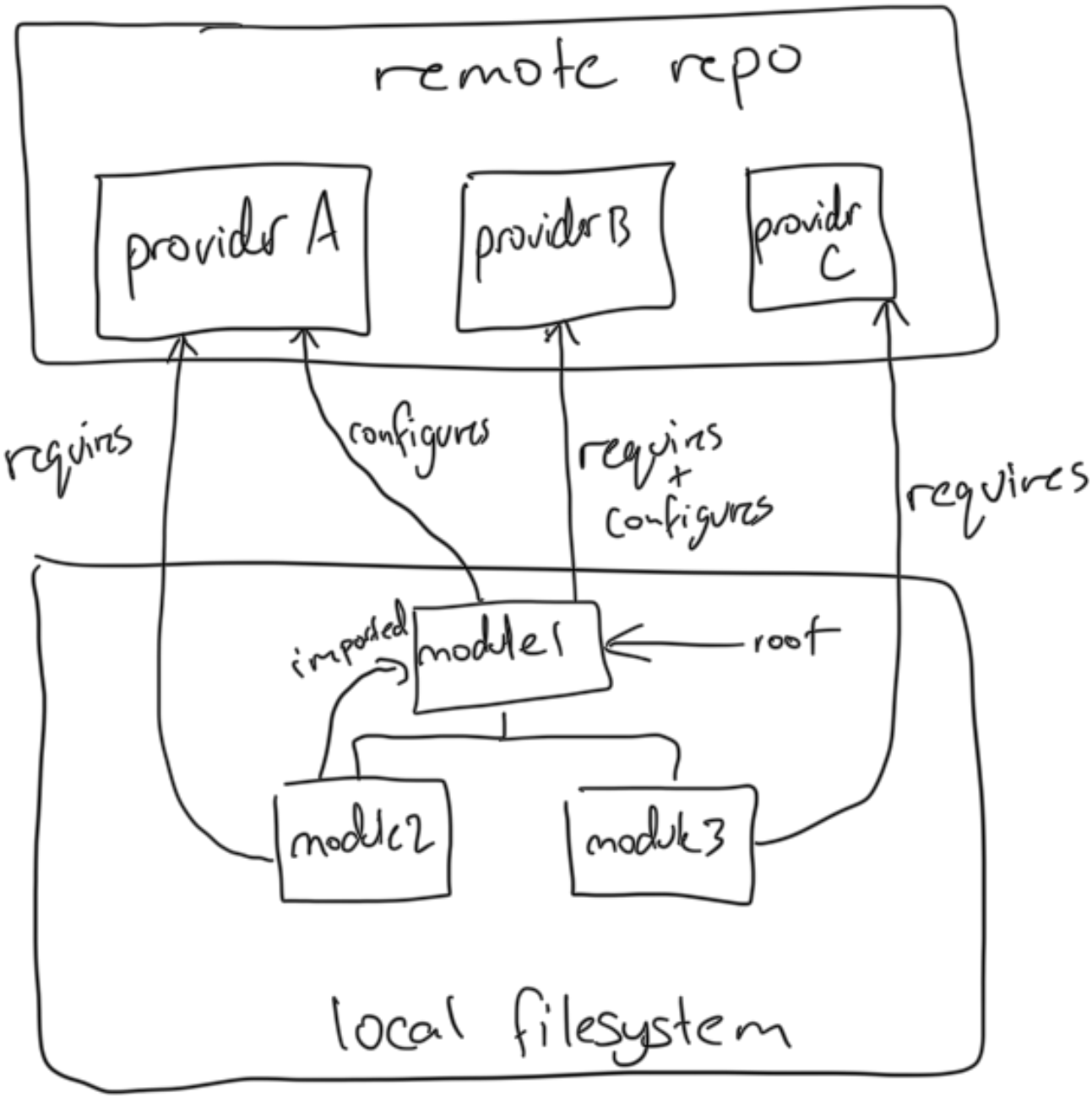
Despite forming a directory tree, modules in subdirectories (i.e. submodules) are not automatically available to modules in parent directories, and modules must import each other to share functionality via a `module` statement in their source files. Local path references allow modules anywhere on disk to be used, and references may be relative or absolute (but are virtually always relative in practice in for code portability). Imports may also reference a remote addresses, and the Tofu CLI will automatically fetch the referenced modules during interpretation. Source addresses can use a variety of protocols, and generally follow the form $protocol$hostname/$namespace/$name/$provider, but note that $provider is just a string. The protocol component is complex, and the source address document has full details, but in general it can be elided for plain HTTP.

Imported modules may have themselves have imports, therefore module dependencies create a directed acyclic graph. The image below illustrates an example with three remote modules (1, 2, 3) and four local modules (A, B, C, D), where B and C are subdirectories of A. Module 1 references module 2; module B references module 1; module A references modules B and 3; and module D references module 3. With this setup the transitive closure of A includes modules 1, 2, 3, and B, but not modules C or D.



Modules may also depend on providers using the `required_provider` statement, which accepts the provider as an argument. Any module may declare a dependency on a provider via a `required_provider` statement, including both the root module and submodules, but the root module alone is responsible for configuring providers using the `provider` statement. All required providers must be configured for successful execution, therefore a degree of coordination between modules is required. Note: Providers do not have transitive dependencies (unlike modules) and therefore do not form a directed acyclic dependency graph.

The image below illustrates an example with three local modules (1, 2, 3) and three providers (A, B, C), where module 1 is the root, module 1 imports module 2, module 1 requires provider B, module 2 requires provider A, and module 3 requires provider C. The transitive closure of providers for module 1 is A and B, therefore it configures both, despite only directly requiring B. Since module 1 does not import module 3, it does not configure provider C.



Modules may also read arbitrary files from disk using the file() and templatefile() functions, and calls may use runtime variables to specify the file path. While this is not best practice, modules can use these functions to read files outside their directory, which has implications for Bazel integration. Any rearranging or renaming of directories and files from external modules could break runtime by creating file-not-found errors, therefore the ruleset must ensure the directory structure of every external dependency is retained at runtime.

Tofu State

Tofu is stateful by design and cannot function without state. It uses state during provisioning for operations that cannot be performed idempotently, and records details that are required for future provisioning operations. State is generally stored on a server for shared access and secret management (as state can contain secrets), but it can also be stored locally in a file. There are multiple ways to specify where state is stored when running workflows with the Tofu CLI, including CLI args and values in the configuration files themselves.

Tofu CLI

The Tofu CLI is the general purpose utility in the Tofu ecosystem and enables several critical workflows. Filtering by those relevant to the ruleset:

1. Provisioning Infrastructure.
2. Validation Sources.
3. Running Tests.
4. Fetching dependencies.

All operations use the current working directory as the root module.

Provisioning Infrastructure

This process turns a Tofu configuration into provisioned infrastructure. It involves three commands:

1. Init, which initialises the working directory and downloads remote modules/providers.
2. Plan, which generates a sequence of operations to reach the desired configuration.
3. Apply, which runs the plan to provision infrastructure (may

Apply prompts the user for approval by default, but this can be disabled with the `-auto-approve` flag.

Validating Sources

The Tofu validate and fmt (format) commands are similar but distinct. Validate checks for syntactic and structural errors in a configuration, whereas format rewrites source files for consistency. Linting is implemented by a separate utility (tflint) and is beyond the scope of this document.

Running Tests

The Tofu test command executes an instrumentation test on a module. It searches for tests, runs them, and prints the results to the console. It will automatically detect tests in the working directory and its immediate “tests” subdirectory, and a specific directory can be specified using the `-test-directory` arg. The working directory must be the module under test, but the module under test does not need to be a fully formed root module, meaning it can have un-configured providers.

Fetching Dependencies

Downloading dependencies occurs as a transparent step during the provisioning workflow, but it can be performed separately to fetch dependencies for other purposes, which is particularly useful for Bazel. By creating a dummy configuration with module/provider dependencies then running `init`, all transitive dependencies will be downloaded to a .terraform directory as follows:

- Modules will be stored in the modules subdirectory, with a separate subdirectory for each module. A generated JSON file maps directories to sources (as source addresses).
- Providers will be stored in the providers directory using the packed directory structure.

TODO need to work out how credentials work

Complexities For Bazel

The Tofu ecosystem is difficult to integrate with Bazel for a variety of reasons:

- Tofu source files are interpreted not compiled and Tofu does not have a native archive format for encapsulating libraries, which makes defining build rules difficult.
- Tofu source files contain both primary source content and package manager directives, and Tofu dependencies are normally downloaded at runtime, which makes build-time resolution of dependencies difficult.
- Tofu dependencies are downloaded using a combination of standard and custom protocols and authentication schemes, which eliminates the use of the standard Bazel downloader/credential-manager.

There are other complexities but these are a few of the main ones. Together they make it difficult to resolve dependencies at build time, package sources and dependencies together, and use the packaged dependencies at runtime.

Requirements

This section outlines granular requirements which translate the overall goal of Bazel-Tofu integration into more specific/actionable targets. All requirements are stated in the form of user stories to ensure each is connected to a tangible benefit and is a meaningful requirement. For comprehension, requirements are divided between build-time, runtime, and other.

In the context of these requirements, an executable Tofu binary is defined as a self-contained executable which wraps the Tofu CLI and runs a single configuration. The executable must package all

dependent modules, providers, and configurations, such that it is effective decoupled from the target platform and external package managers. This definition was selected to reduce ambiguity in the requirements and allow pragmatic design without strictly defining the solution.

The following requirements are specific to build-time (including CI). As a ruleset user:

1. I want to specify external dependencies in one place, so I can audit, review, and maintain all external dependencies.
2. I want to specify external dependencies using every source address type supported by the Tofu CLI, so I have access to all deps supported by native Tofu.
3. I want to pin external dependencies, so I can be alerted if an external dependency changes unexpectedly.
4. I want to enforce a single-version policy for all external dependencies by default, so I can be intentional about versions and maintain engineering discipline.
5. I want to allow single-version policy overrides for individual external dependencies, so I can use external dependencies that are incompatible with a single-version policy but only by exception.
6. I want to build reusable Tofu modules, so I can use follow general engineering best practices and principles (including composition, DRY, SRP and more).
7. I want to validate and lint reusable Tofu modules, so I can check they are syntactically and semantically correct.
8. I want to test reusable Tofu modules, so I can check they are functionally correct.
9. I want to build an executable Tofu binary from a Tofu module, so I can provision infrastructure with hermetic and reproducibly.
10. I want to build an executable Tofu binary from a Tofu module, so I can sign the artefact and guarantee runtime integrity.
11. I want to build an executable Tofu binary from a Tofu module, so I reduce runtime dependence on the host system and remote package managers.
12. I want my builds to scale to arbitrary targets sizes, so my repository structure is not limited by the ruleset.
13. I want actionable and precise error messages, so I can debug effectively.
14. I want builds to be hermetic and reproducible, so I can use Bazel as intended by Google.
15. I want to execute builds on MacOS, Linux and Windows, so I can develop on virtually any machine.
16. I want to use Bzlmod, so I can modern Bazel workflows and tools.

The following requirements are specific to runtime of an executable Tofu binary. As a ruleset user:

17. I want to execute binaries on MacOS, Linux and Windows, so I can provision and interact with infrastructure on virtually any machine.
18. I want to perform any valid Tofu CLI function via the executable binary and have it apply to the packaged module, so I can perform advanced workflows.

19. I want the default workflow to be plan/apply with interaction prompts, so I can perform basic workflows easily and safely (i.e. avoid accidental infrastructure changes).
20. I want to pass through any valid Tofu CLI command to an executable Tofu binary and have it operate on the packaged Tofu module, so I can perform advanced infrastructure management.
21. I want actionable and precise error messages, so I can debug effectively.

The following requirements are not related to build-time or runtime. As a ruleset user:

22. I want extensive documentation, so I can understand and use the tools with confidence.

A few notes on design choices:

1. WORKSPACE support is not required as it will be deprecated in the next version of Bazel (9).

Solution

The solution is presented as concept, architecture, and implementation.

Concept

The ruleset allows users to treat Tofu as a compiled language by producing library and binary artefacts that bundle first party components with their transitive dependencies, and in the case of binaries, a runtime. The ruleset achieves this by downloading external dependencies at build-time instead of runtime, creating structured archives for compilation output, and providing a runtime that can execute an archive to provision architecture. It involves nuance and complexity, but in general, the ruleset can be considered as a way to turn interpreted Tofu sources into compiled artefacts and execute them at runtime.

TODO images explaining the above.

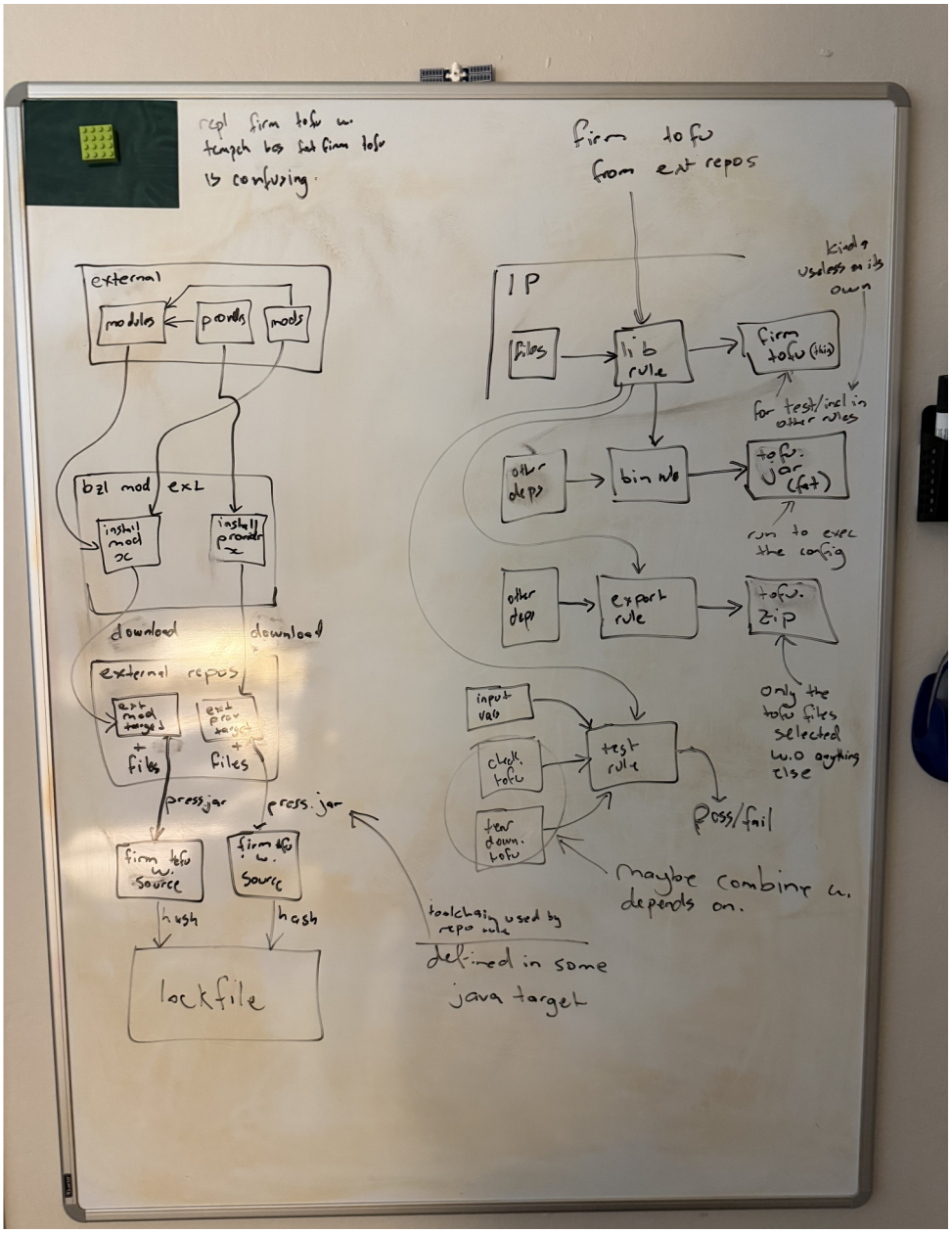
For users of other rulesets, such as rules_java and rules_jvm_external, the process of using rules_tofu is directly analogous and will be familiar. In rules_tofu, external dependencies are declared in a MODULE.bzl file using a module extension (just like maven deps) and artefacts are defined with build rules (just like java_library, java_binary and java_test). External and internal dependencies are managed with dep attributes in the build rules instead of package manager statements in the source files. This approach changes the build focus from manual package management to declarative composition.

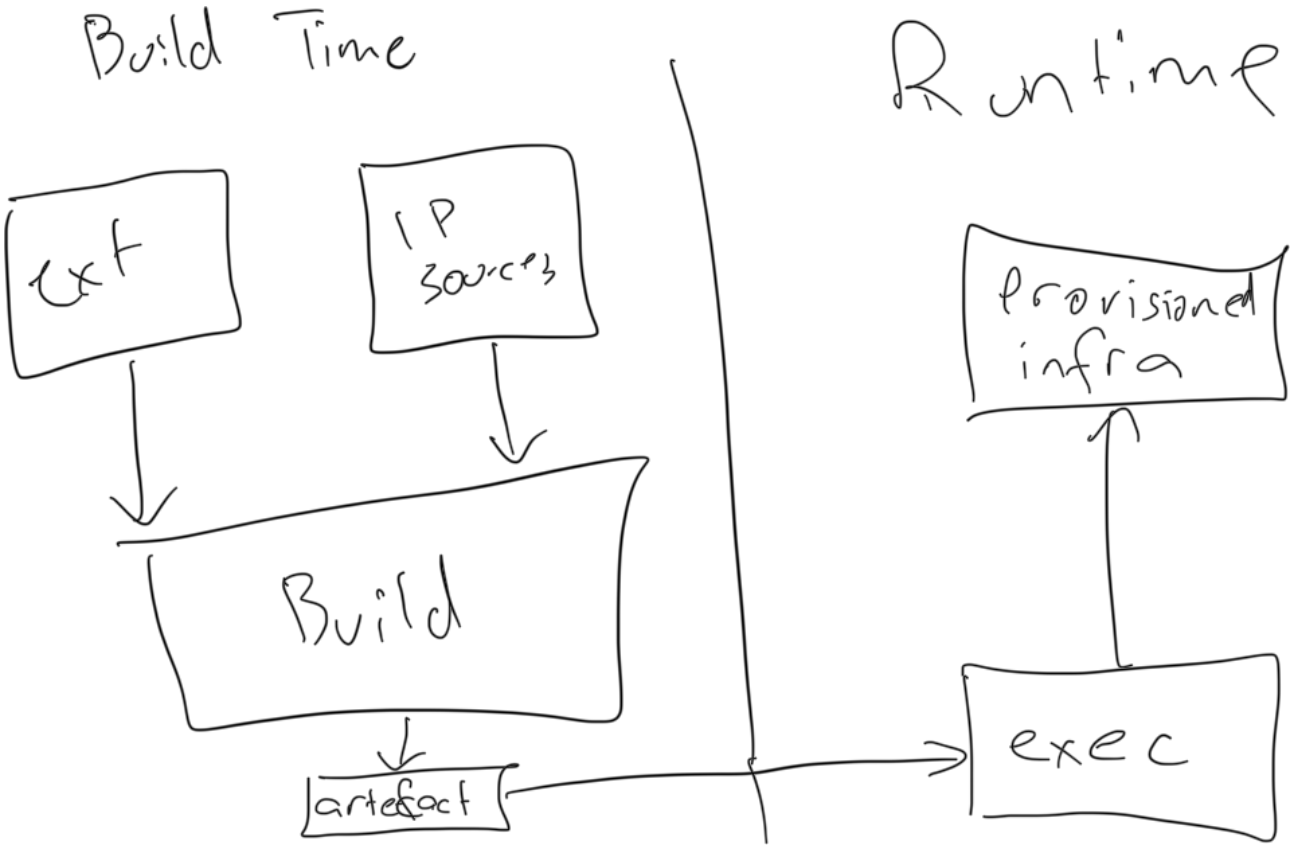
Aspect	Native Tofu	Bazel Tofu
External Dependency Resolution	Runtime	Build-time
External Dependencies Defined	In source files	In build-system files
Execution Model	Interpreted	Compiled (interpretation abstracted)
Library Artefact	Directory tree of sources	Single archive object
Executable Artefact	Directory tree of sources	Single executable object
Test Artefact	Directory tree of sources	No well-defined artefact,

		handled by Bazel infrastructure
Build Task	Navigate to source directory and validate sources with Tofu CLI (only available for root modules)	Run bazel build //foo:lib (available for root and non-root modules).
Run Task	Navigate to source directory and plan/apply sources with Tofu CLI	Run bazel run //foo:bin
Test Task	Navigate to source directory and plan/apply sources with Tofu CLI	Run bazel test //foo:test
Runtime	Tofu CLI installed on target machine	JRE installed on target machine
Module Composition System	Reference other modules in sources via their local filesystem path or remote source address	Declare Bazel dependencies and reference in sources with their Bazel path.

Overall this system leans heavily into established Bazel paradigms to allow Tofu programs to benefit from the same engineering discipline and best practices that are taken for granted in other languages. With rules_tofu, engineers and developers can rest assured their builds are hermetic and reproducible, and more easily manage runtime deployment and execution.

In essence, the ruleset shifts dependency management to build-time, handles compiling sources/deps into single archive files, abstracts interpretation, and decouples program composition from the underlying filesystem. With this ruleset, users can reason about Tofu programs as compiled, composable, testable, deployable, units, just as they would any other language.


repl jvm tofu vs. transpch bzl test jvm tofu is confusing
jvm tofu from ext repos
external
modules
providers
models
bzl mod ext
install module rc
install provider rc
download
download
external repos
ext mod target
+ files
ext prov target
+ files
jvmjar
jvmjar
jvm tofu w. source
jvm tofu w. source
hash
hash
lockfile
IP
files
lib rule
jvm tofu (this)
kinda useless on its own
for test/incl in other rules
other deps
bin rule
tofu: jar (fat)
run to exec the config
other deps
export rule
tofu: zip
only the tofu files selected w.o anything else
input vals
check tofu
test rule
pass/fail
tear down tofu
maybe combine w. depends on.
toolchain used by repo rule
defined in some java target



Architecture

The ruleset can be explained in terms of component and processes. The components are discrete elements that perform specific functions, and the processes are the workflows they create when combined as intended. In other words, components are specific objects, processes are emergent behaviours.

The components of the ruleset are:

1. FirmTofu Archive, a file which packages Tofu programs for composition and execution.
2. TofuPress, a program which turns Tofu modules and providers into single archive objects.
3. Soybean, a program which downloads external dependencies.
4. Vegan, a program which unpacks and executes FirmTofu archives.
5. BakedTofu, a program which performs infrastructure provisioning in one step.
6. Library Rule, a Bazel build rule which turns sources into libraries.
7. Import Rule, a Bazel build rule which imports pre-built archives.
8. Export Rule, a Bazel build rule which exports libraries for external package managers.
9. Binary Macro, a Bazel build macro which turns libraries into executables.
10. Test Rule, a Bazel build rule which runs tests on libraries.
11. Tofu CLI Repository Rule, a Bazel repository rule to fetch the Tofu CLI.
12. Tofu Dependency Repository Rule, a bazel repository rule to fetch external Tofu dependencies.
13. Module Extension, a Bazel module extension for configuring the ruleset.

14. Toolchains, a series of Bazel toolchains to support the build rules/macros.

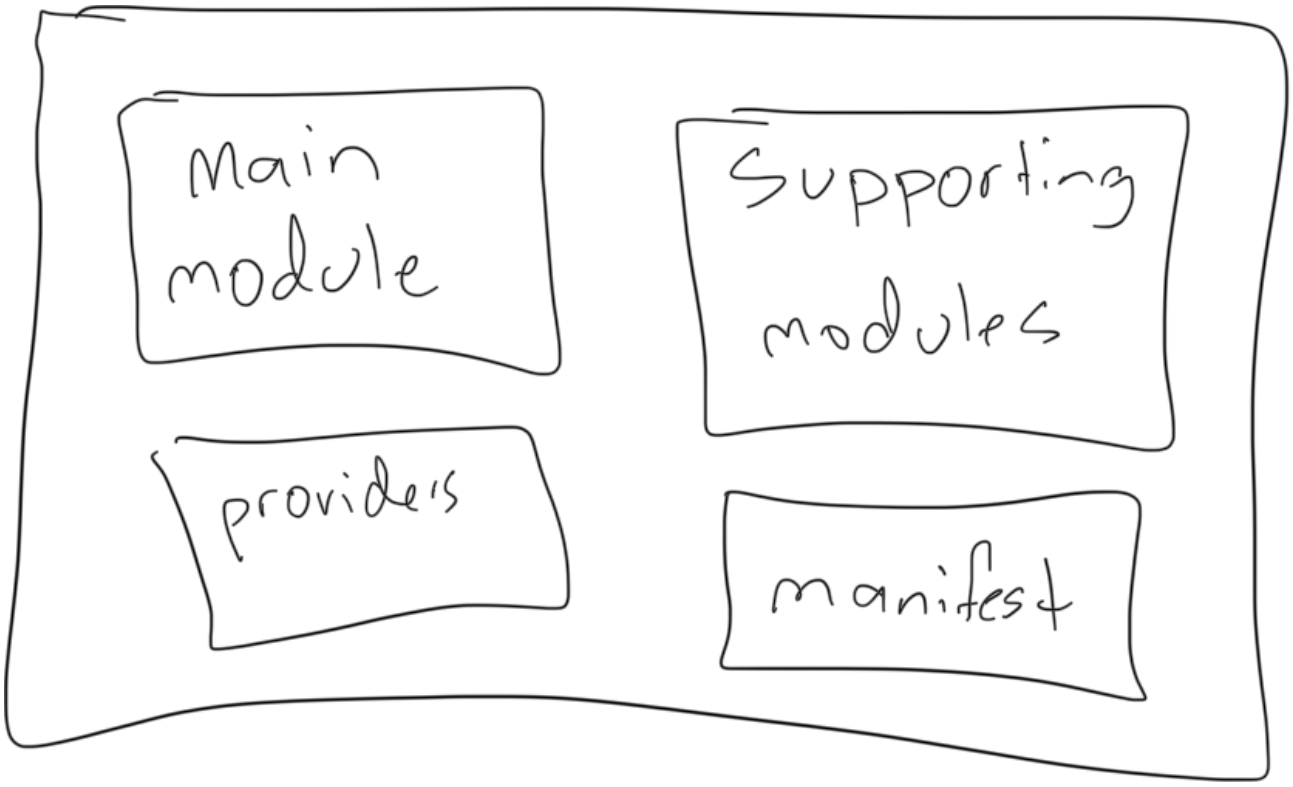
The processes of the ruleset are:

1. Dependency Management, which involves resolving and downloading external dependencies for use in build execution.
2. Build Execution, which involves using first party sources and external dependencies to produce an executable.
3. Runtime Execution, which involves running an executable to provision infrastructure.

Each listed component/process is introduced with enough detail to explain what it is, what it does, and why it exists. The more complex items are further detailed in the Implementation section.

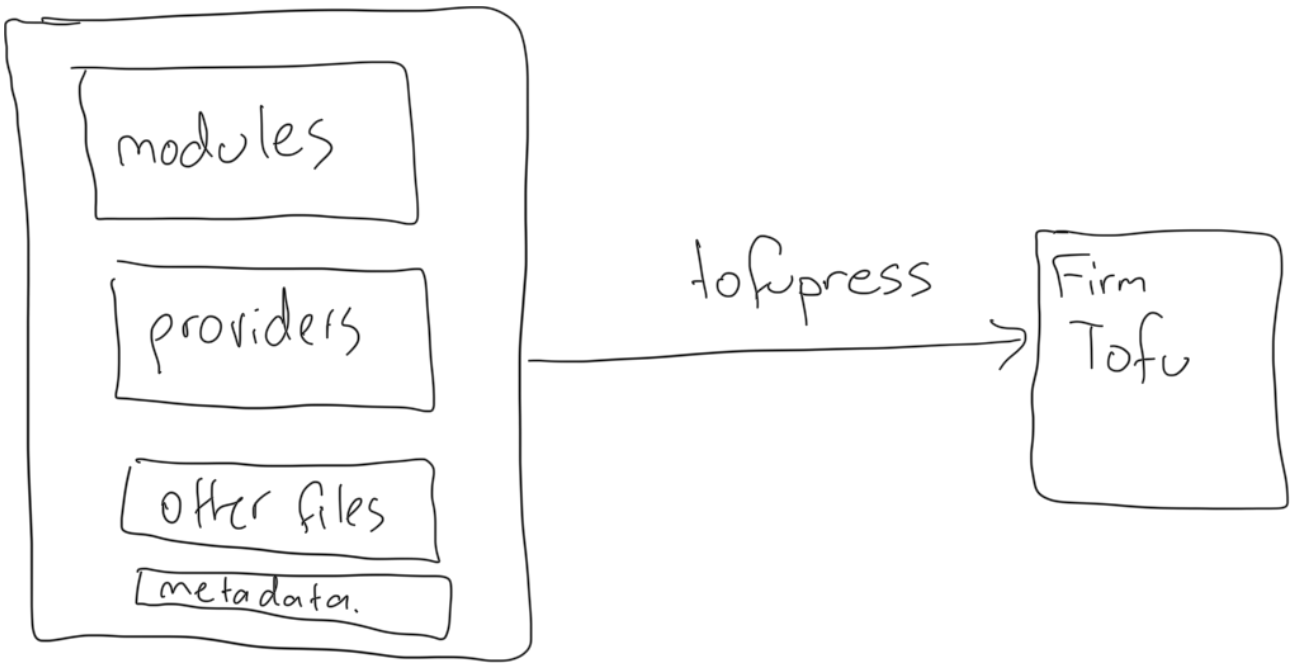
FirmTofu Archive

The ruleset provides the FirmTofu archive format, which is effectively a zip containing a manifest, the main module for a FirmTofu program, and its transitive dependencies (modules and providers). It was created to allow transitive dependencies to be resolved at build-time and packaged with first party sources, and was designed to enable performant and effective build operations (archives can be easily merged in linear time).



TofuPress

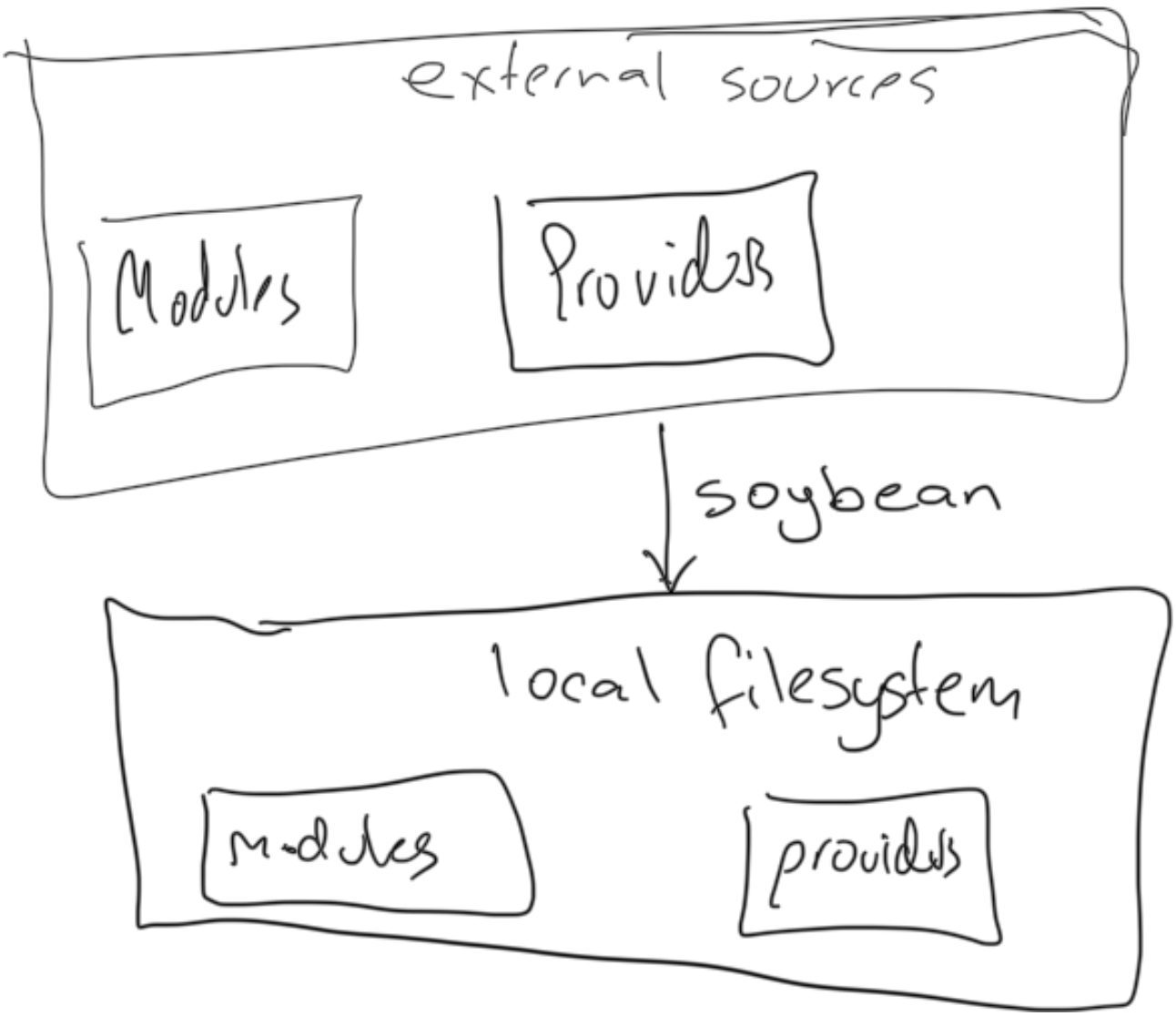
TofuPress is a custom Java program for compiling Tofu modules and providers into FirmTofu archives so they can be distributed and used across the build graph (amongst other supporting functions). It was implemented in a high-level language and kept separate from Starlark to allow usage outside Bazel. A useful mnemonic is “TofuPress turns Tofu into FirmTofu”.



Note: In previous iterations of the design, TofuPress was also responsible for downloading dependencies, unpacking archives at runtime, and executing their contents. These responsibilities were split out into the Soybean and Vegan programs to better follow the SRP, reduce unnecessary complexity internally, and offer a more granular toolset to the user. TofuPress still uses the Vegan runtime internally for various functions (more details provided in implementation).

Soybean

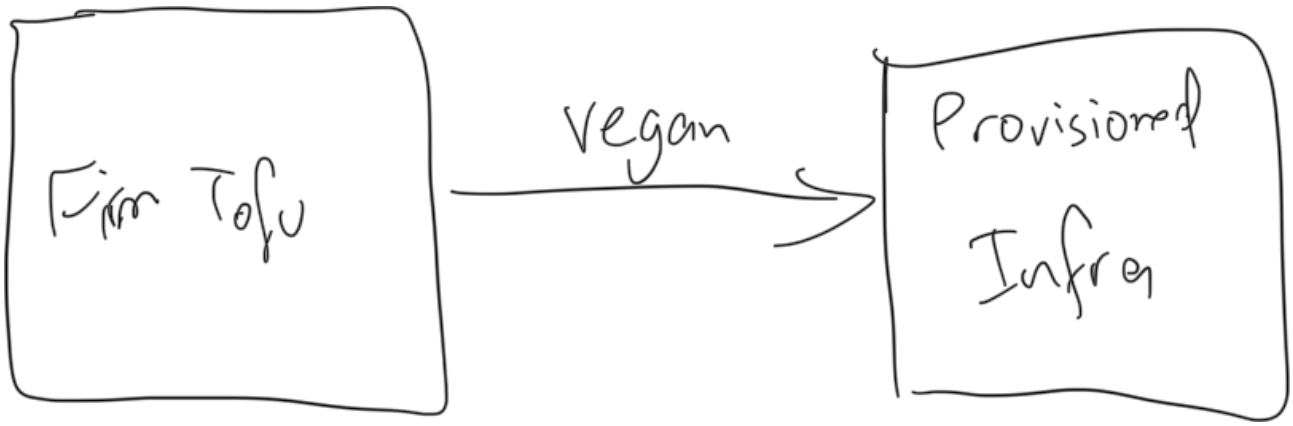
Soybean is a custom Java program for resolving external dependencies and downloading them to the local filesystem (amongst other supporting functions). It allows the ruleset to shift dependency resolution from runtime to build-time. It was implemented in a high-level language and kept separate from Starlark for separation of concerns and ease of testing. A useful mnemonic is “Tofu comes from Soybean”.



Note: Soybean does not fetch the Tofu CLI. That is handled by a repository rule (see below).

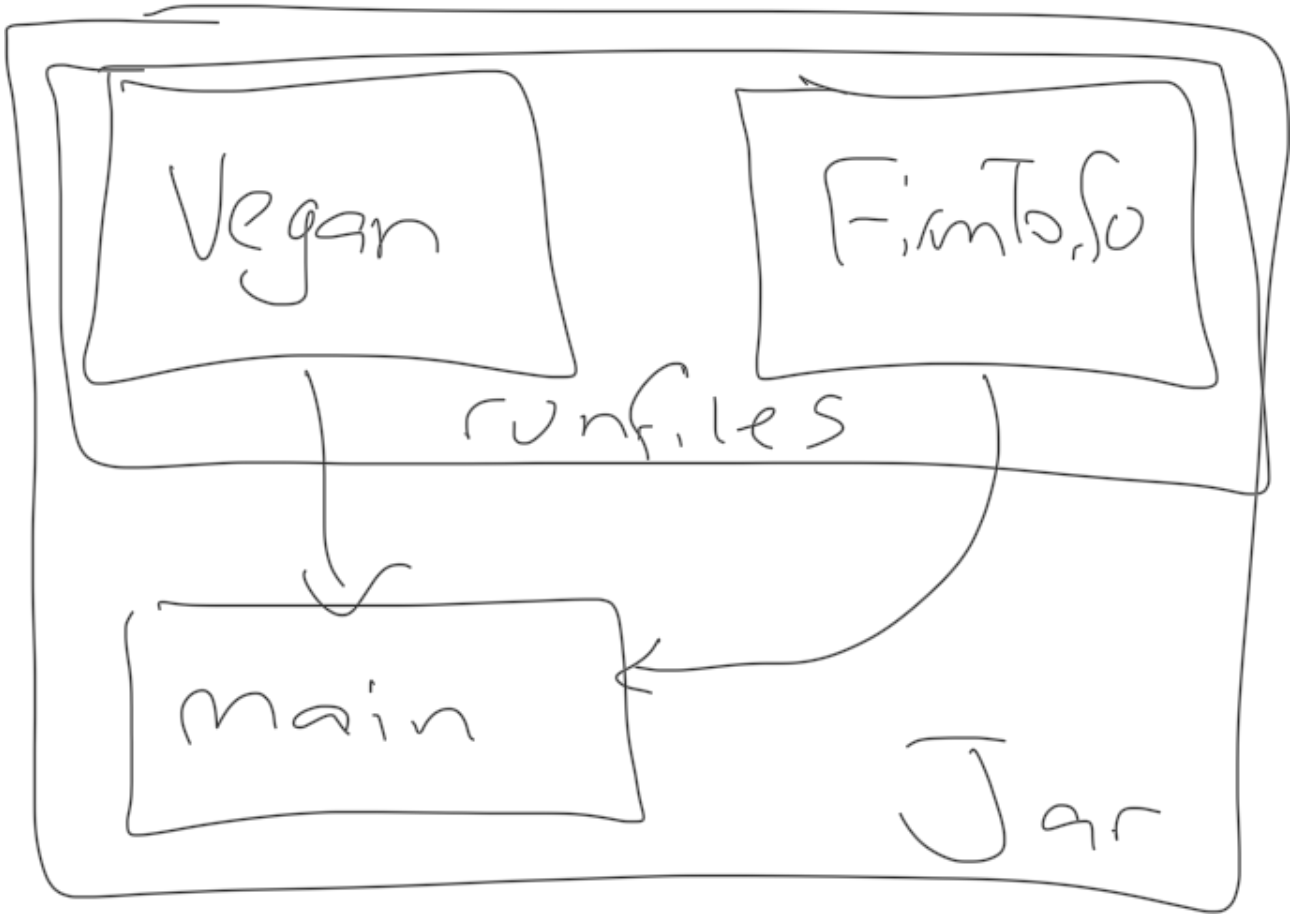
Vegan

Vegan is a custom Java which functions as a runtime for FirmTofu archives. It orchestrates a complex sequence of unpacking, linking, and interpretation, but abstracts away the details so the user can reason about Vegan as simply consuming a given FirmTofu archive to provision infrastructure based on its contents. It was implemented in a high-level language and kept separate from Starlark because its needed at both build-time and runtime. A useful mnemonic is “Vegan consumes FirmTofu at runtime”.



BakedTofu Executable

The ruleset provides the BakedTofu executable format, which is effectively a Java JAR containing the Vegan runtime, a FirmTofu archive, and a main program to link them together. It exists to simplify execution at runtime so the end user can reason about provisioning as running a single executable. It's not single program but rather a class of programs, as it comes form the Binary Rule and each instance bundles different data.

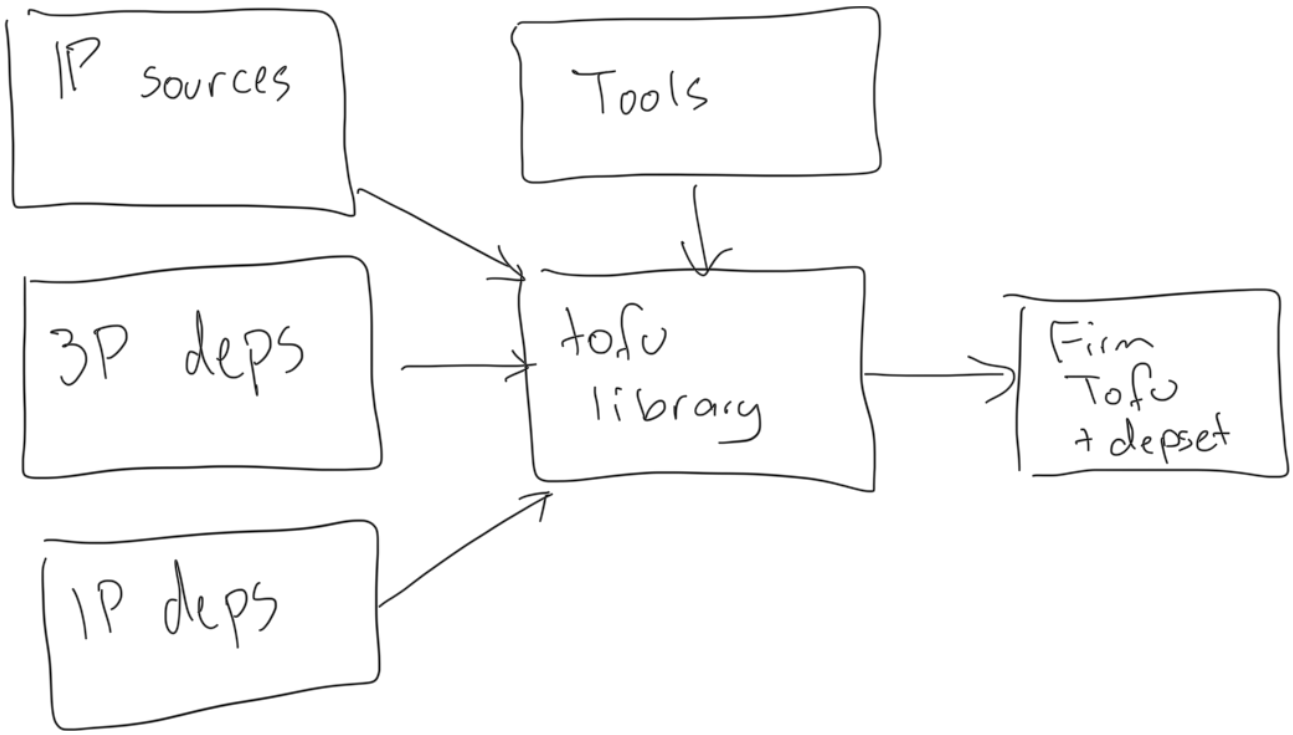


BakedTofu executables are produced by the tofu_binary build macro (introduced below).

Library Rule

The library build rule (tofu_library) is analogous to java_library. It accepts sources and dependencies, and compiles them into a composable library object (FirmTofu). It accepts sources and dependencies and produces a FirmTofu archive and a depset (for performance optimisation).

TODO redraw all rule/macros as functions not objects



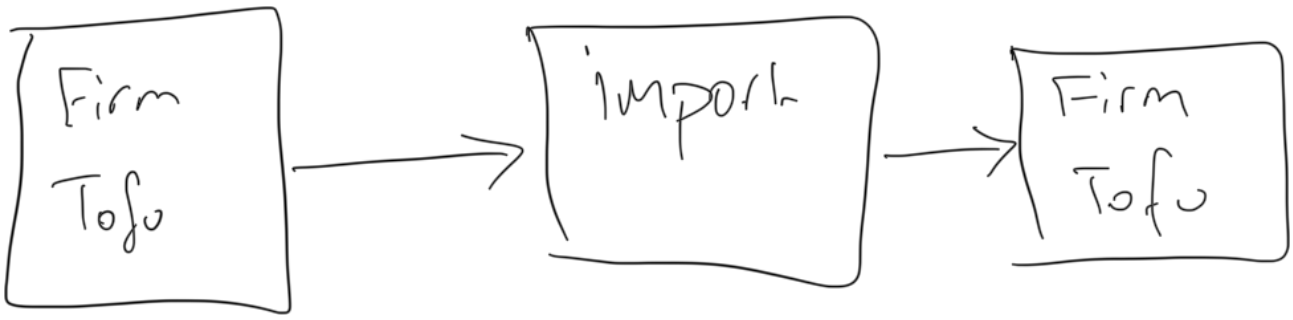
TODO need to add metadata to image above

Export Rule

The export rule (tofu_export) is analogous to java_export. It accepts sources and dependencies, and compiles them into a composable library object (FirmTofu). Unlike the library rule, the output includes all transitive dependencies, and is well suited to release to external package managers.

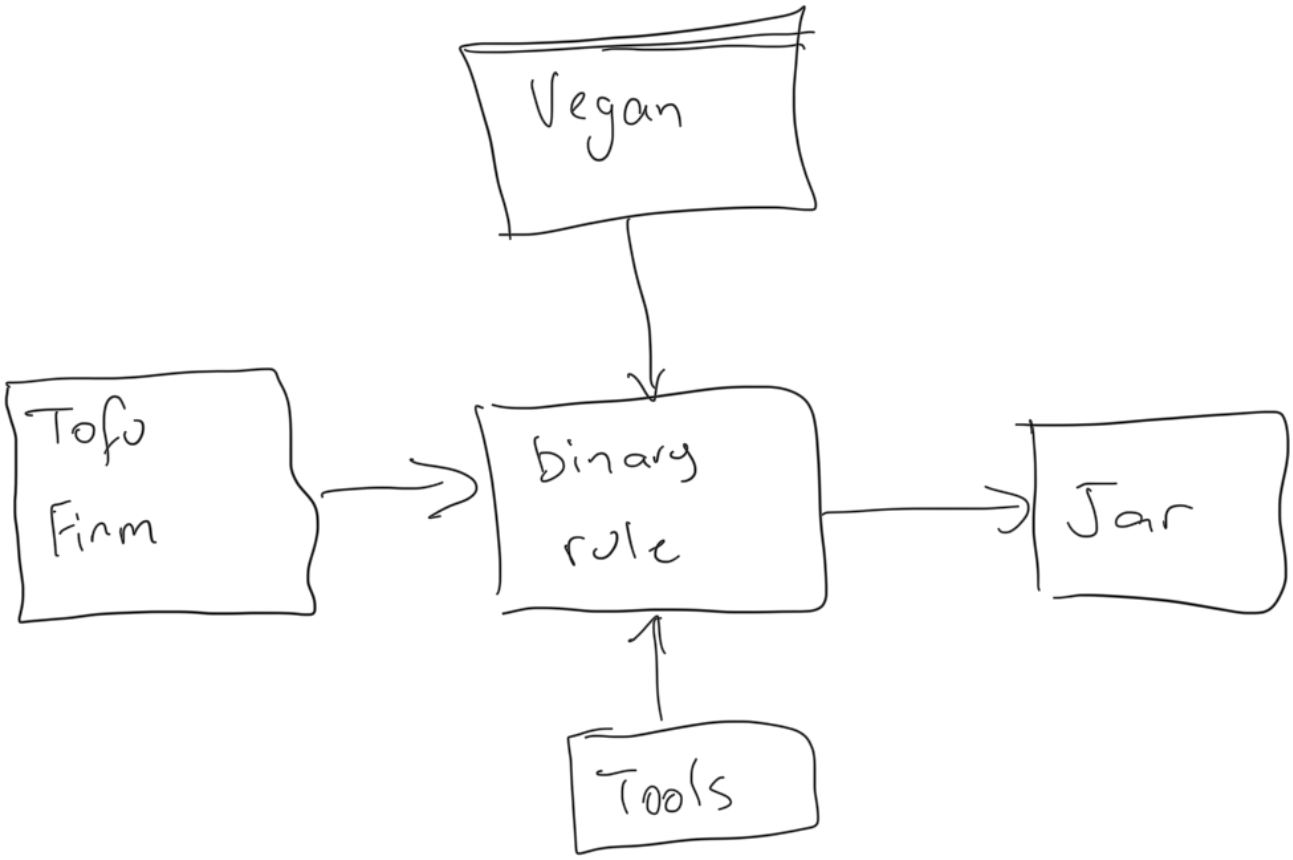
Import Rule

The import rule (tofu_import) is analogous to java_import. It accepts a FirmTofu file and makes it available to the build graph. It's functionally the simplest of all the build rules and is effectively just a pass through to integrate pre-compiled archives.



Binary Macro

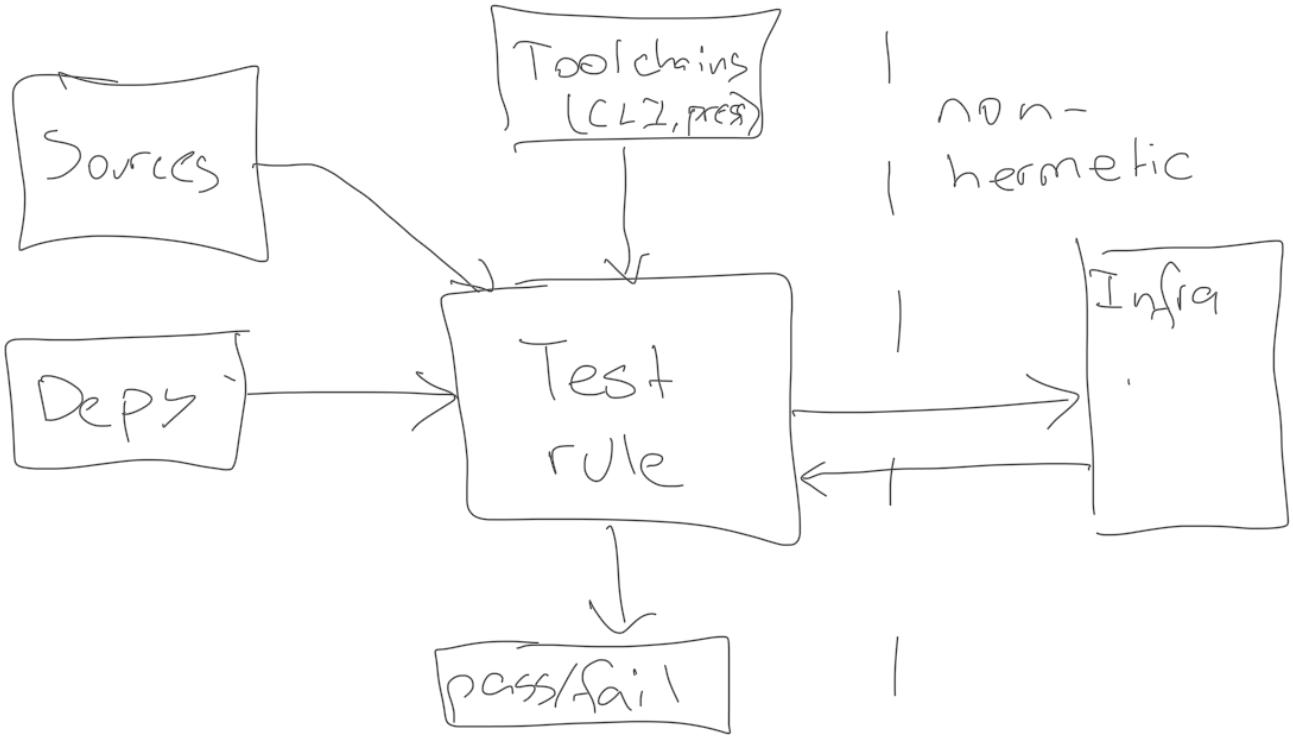
The binary macro (tofu_binary) is analagous to java_binary, and actually uses java_binary internally to produce an executable. It accepts a Tofu library, and bundles it with the Vegan runtime and a minimal program that defers to Vegan when run. By delegating to the Vegan runtime, the executable can unpack and execute the bundled FirmTofu object. This rule can be used to produce self-contained executable that provision infrastructure.



Test Rule

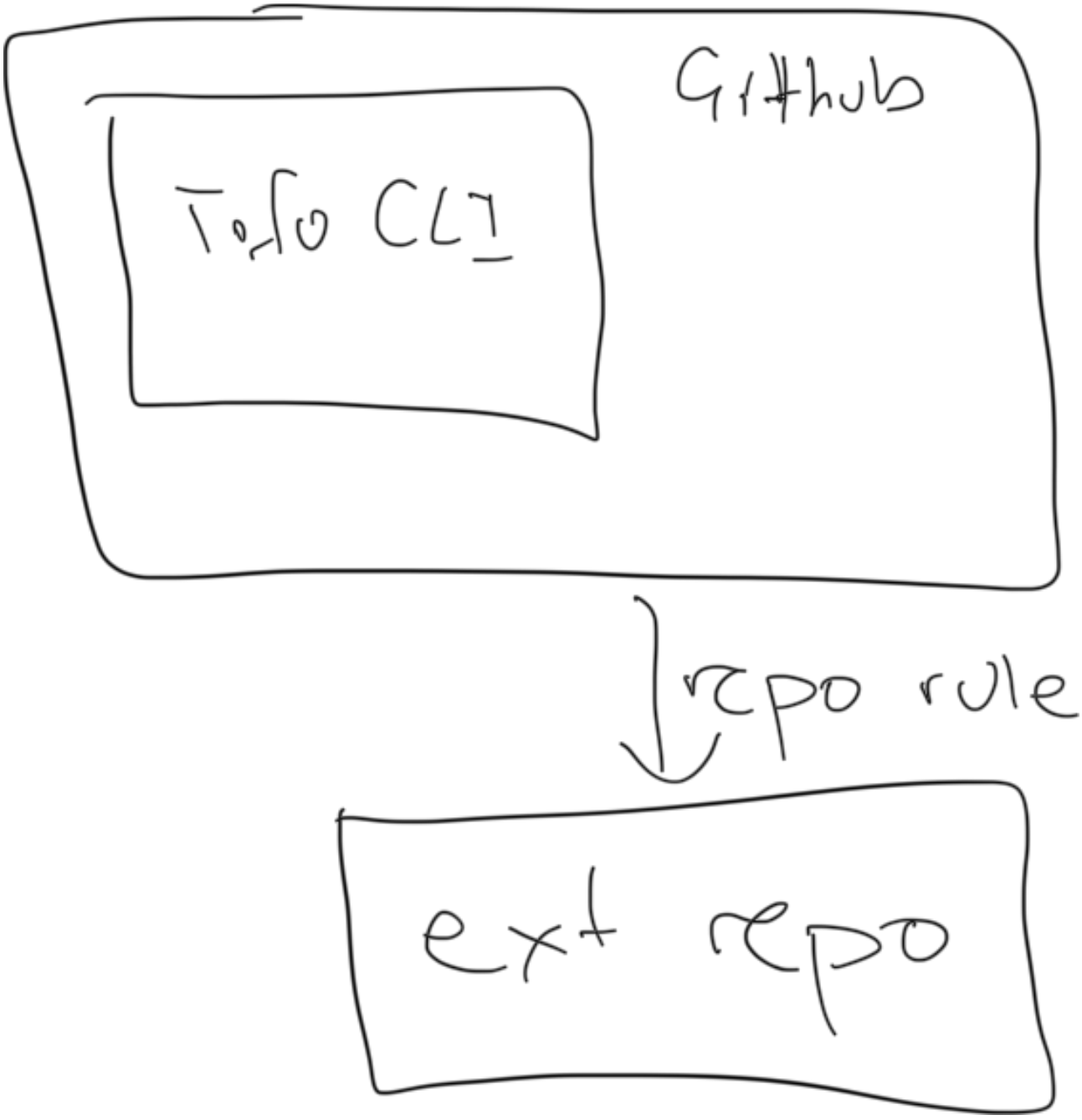
The test rule (tofu_instrumentation_test) does not have a direct Java analogue because it is an instrumented test not a unit test. This limitation emerges from the ecosystem which does not have a concept of local testing and is entirely reliant on provisioning real infrastructure to determine program correctness. The rule simplifies testing by accepting the library under test as a dependency and using the specified sources to setup the text fixtures (e.g. set variables, perform checks, etc). It works well

when libraries follow general engineering best practices such as avoiding global/static data so tests con specify their behavior with input variables.



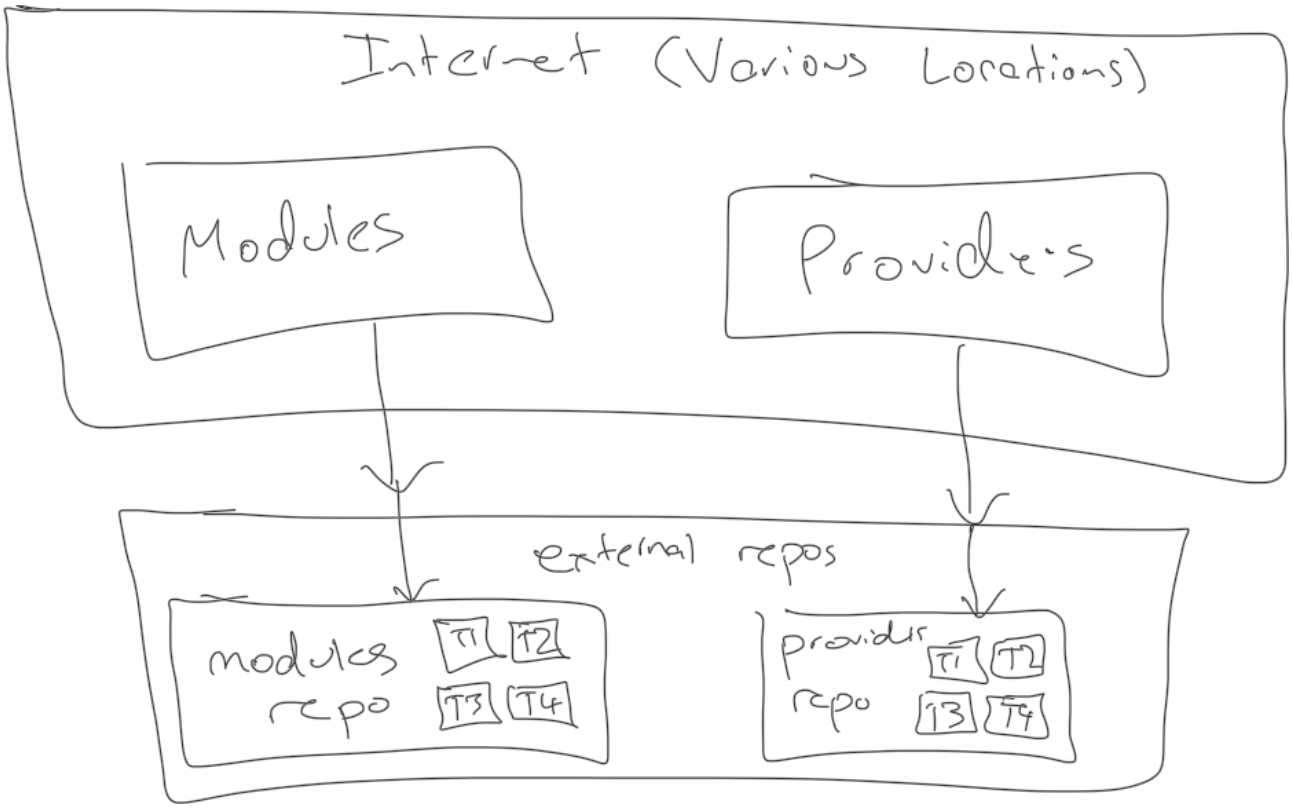
Tofu CLI Repository Rule

The ruleset provides the Tofu CLI repository rule to make the Tofu CLI available to the build. It accepts the address to download from and performs the download using the standard ctx.download operation. It exposes the CLI as an executable file target, and handles the platform resolution steps using the select function.



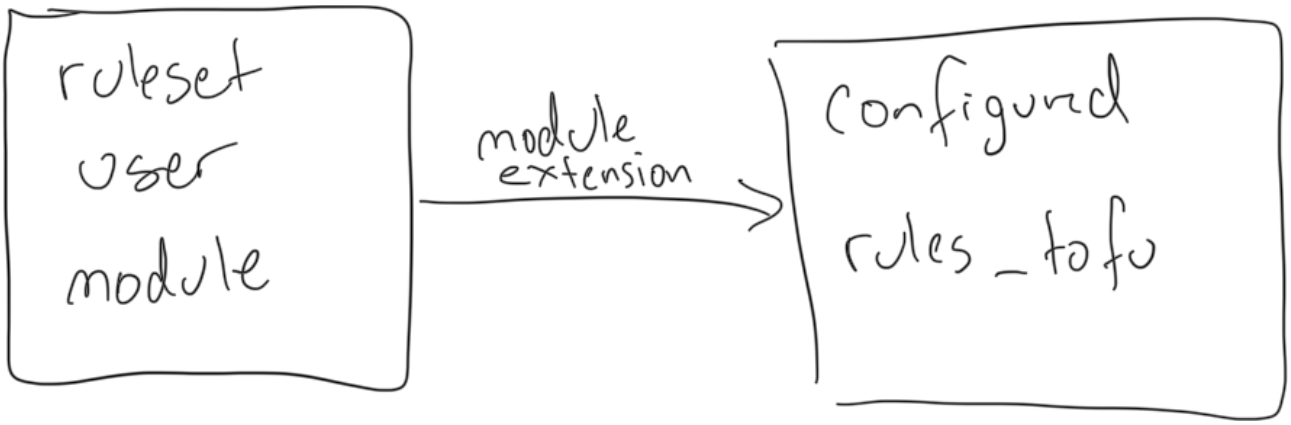
Tofu Dependency Repository Rule

The ruleset provides the tofu dependency repository rule to make external modules and providers available in the build. It accepts a flat list of modules and providers, downloads them using Soybean, and creates targets to make them accessible. In a previous design iteration the rule was divided into two (one for modules one for providers) but they were merged to simplify generation tasks.



Module Extension

The ruleset provides a single module extension for integration with the ruleset user's module. It exists to collect configuration information and orchestrate external dependency resolution via the build rules.



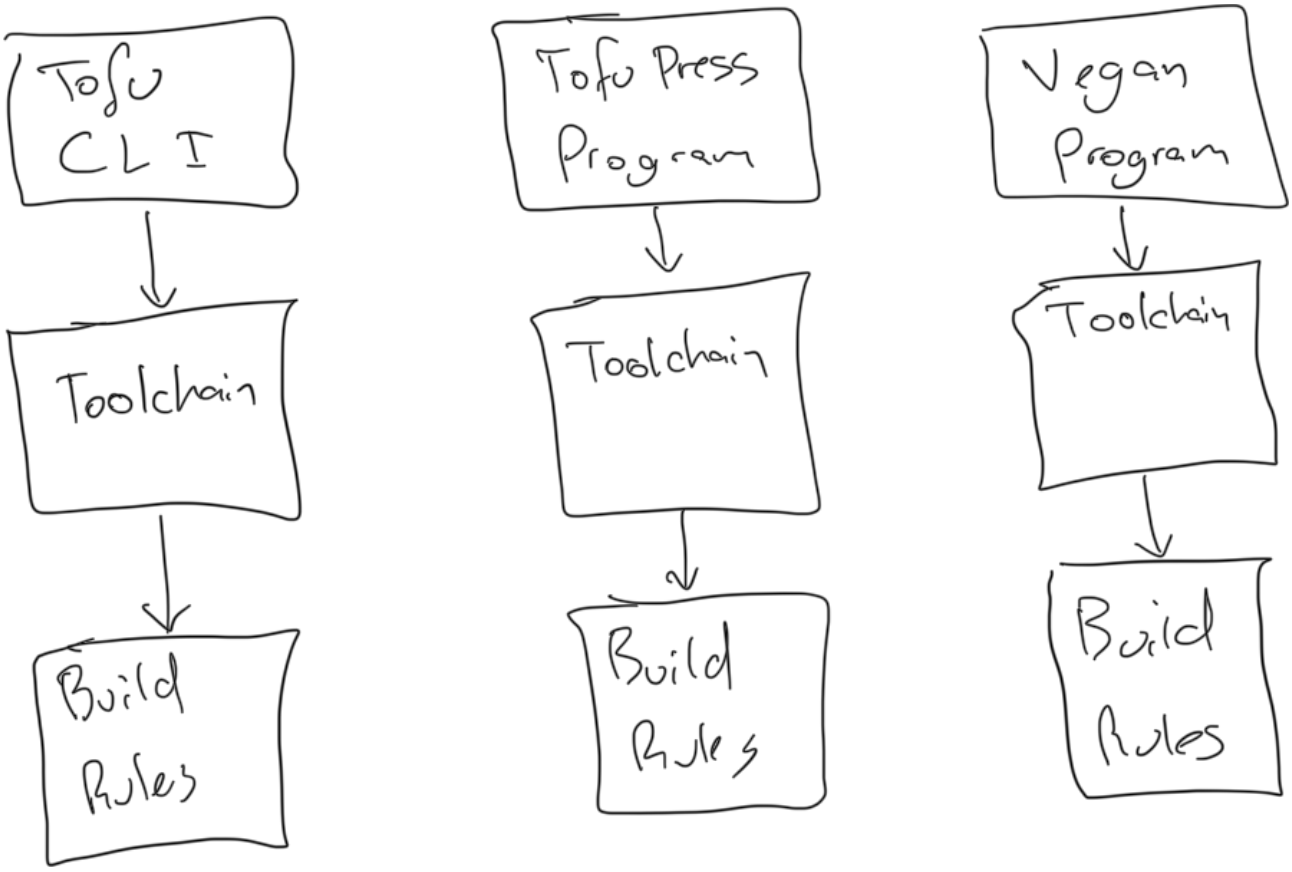
Toolchains

The ruleset provides the following toolchains to support the build operations:

- The Tofu CLI toolchain, which wraps the Tofu CLI.
- The TofuPress toolchain, which wraps the TofuPress program.

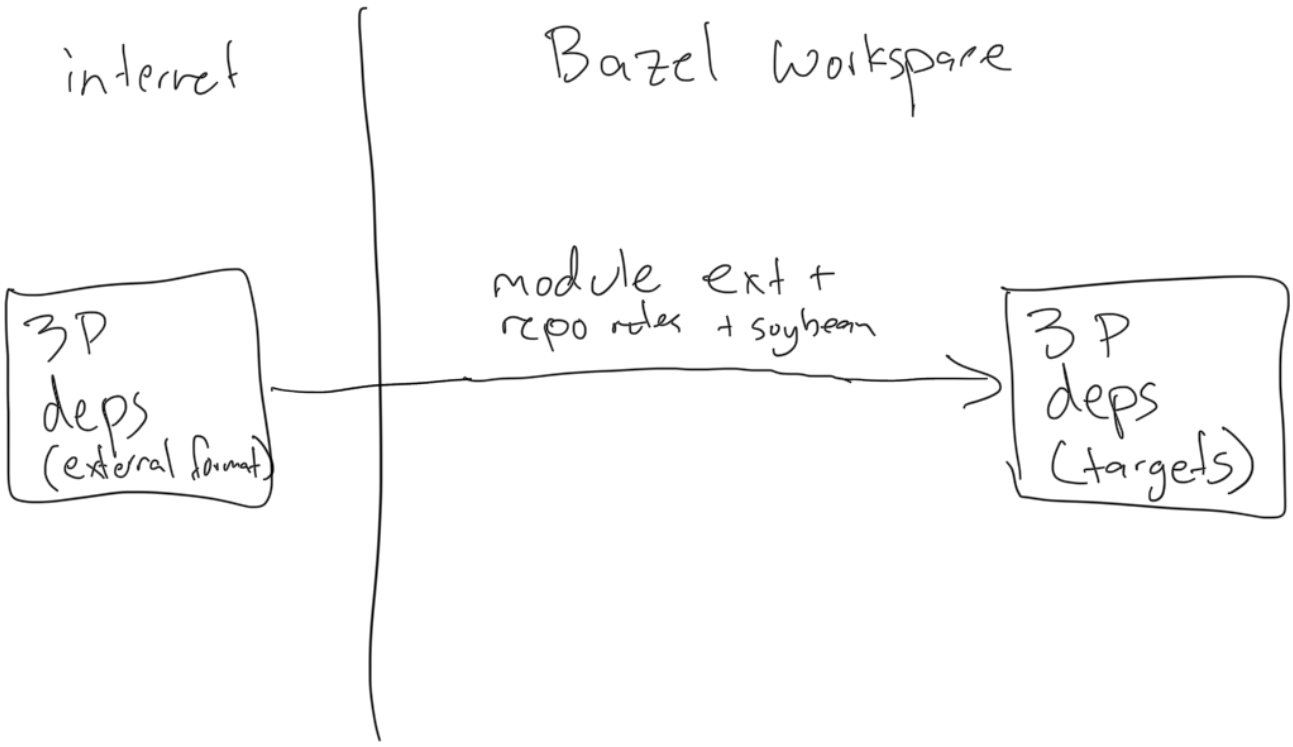
– The Vegan toolchain, which wraps the Vegan program.

No toolchain is provided for Soybean because it is only used by the module extension and repository rules, both of which cannot use toolchains.



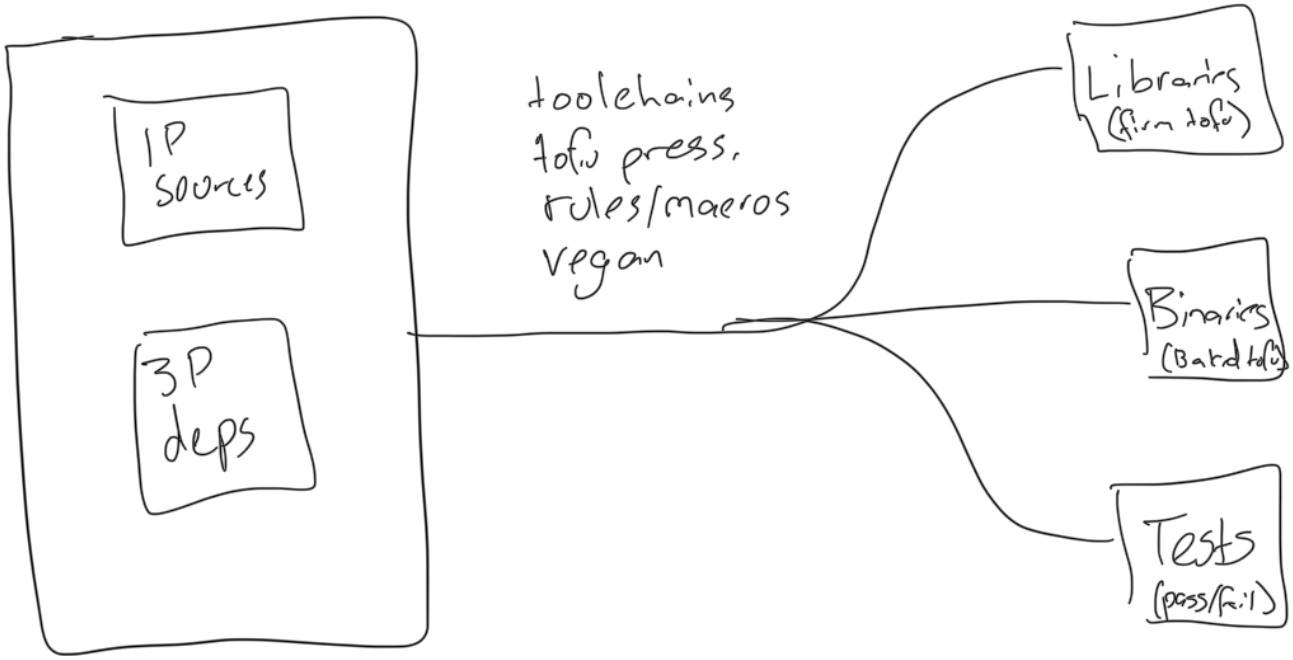
Dependency Management

Dependency management is the process of resolving external dependencies and making them available to the Bazel workspace. It allows ruleset users to reference external dependencies in their build as build targets. It involves the module extension, the repository rules, and the Soybean program. It begins when a dependency is transitively required by the Bazel build graph and halts when all required dependencies has been resolved (or fail to resolve).



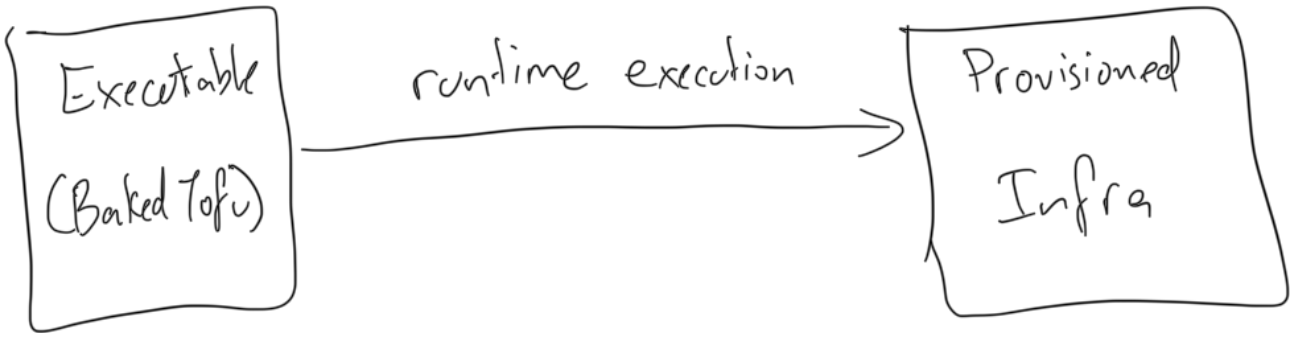
Build Execution

Build execution is build-time process of producing artefacts from sources and dependencies, with the end goal of producing libraries and executables, and running tests It involves the build rules/macros, the TofuPress program, the Vegan program, and the toolchains. It begins when the ruleset user invokes a build operation which involves a Tofu build rule/macro and ends when the build completes (successfully or unsuccessfully).



Runtime Execution

Runtime execution is the process of executing a BakedTofu executable to provision infrastructure. It involves the BakedTofu executable, the FirmTofu archive, and the Vegan runtime. The process begins when a BakedTofu executable is invoked by the end use at runtime an ends when infrastructure provisioning completes (successfully or unsuccessfully).



Implementation

All components and processes are complex enough to warrant further detail, except for the Import Rule and the Tofu CLI Repository Rule which are straightforward and require virtually no design work.

FirmTofu

FirmTofu is an archive format for compiled Tofu programs. It uses zip as the underlying format for broad OS/tooling compatibility, and its contents are divided into three top-level structures:

1. Modules, a directory containing the modules of the program.
2. Providers, a directory containing the providers of the program.
3. Manifest, a protobuf binary wire format file containing metadata about the archive and the program.

The modules/provider directories contain compiled modules/providers alongside a metadata file for each (a Google protobuf version 3 binary wire format file). Both are named using the hashing algorithm defined below, with `.pb` appended to the metadata files. Metadata files may exist without corresponding content directories for cases where only metadata is required, but content directories cannot exist without corresponding metadata files.

This structure creates content addressable storage for modules/providers, which ensures reproducibility across builds, meaning the same set of modules and providers will always produce the exact same archive. Furthermore, it decouples the archive structure from native Tofu, so that upstream changes to native Tofu do not necessarily invalidate existing archives and can be supported by changing the TofuPress program (critical for long-term sustainability).

A critical function of the metadata files is recording the relationships between modules and providers. These details are required at build-time and runtime for performance optimisation and correct operation. Each metadata file contains both the forward links (i.e. items this item references) and the backward links (i.e. items which reference this item). The forward links are retained so content addressable storage can be transformed back to a nested directory tree at runtime (recall from

background section that all files must be placed in their original location), and the backward links ensures performant archive modification when removing modules/providers (discussed below).

A few design choices worth noting:

- Google Protobuf (version 3) was chosen over XML and JSON to ensure files are space-efficient, inherently reproducible, and can be checked against a predefined schema.
- A previous design iteration stored the metadata for each module/provider in the manifest itself, but this was not ideal, as it burdened a single file with the entire contents of the archive and required reading the entire file into memory before making changes.
- A previous design iteration explored further breaking down modules/providers directories into content addressable storage so each file had a unique address. While this may have offered further deduplication across modules, it incurred significant duplication in metadata, and was not pursued. This avoids building a general purpose content addressable storage solution (not the goal) and keeps the format optimised for Tofu programs (modules/providers are the primary unit of reuse).
- While the Bazel ruleset is designed for single-versioning by default, the archive supports multiple versions of modules/providers, which is critical for cases where a single-version cannot satisfy all transitive dependencies.

Overall, this system stores modules and providers in a way that scales to large programs, is resilient to change in the broader ecosystem, and supports the arbitrary file reference system of Tofu programs.

Modules Directory

The modules directory contains the modules of a Tofu program. It contains a metadata file for each module with an optional subdirectory containing its content. Each module content directory may contain any number of arbitrary files but never nested directories.

Below is an example of a modules directory containing three modules, only two of which have content.

```
/modules
  e5a0b54f2c8d2a6a0c8e4f6b2d1c0e8a7f6d4c2b0a9e8d7c6b5a4f3e2d1c0b9a.pb
  3a9850a2665812933f52423193494b923985b428234857342938472394872349.pb
  f0e1d2c3b4a5968778695a4b3c2d1e0f1a2b3c4d5e6f7a8b9c0d1e2f3a4b5c6d.pb
  /e5a0b54f2c8d2a6a0c8e4f6b2d1c0e8a7f6d4c2b0a9e8d7c6b5a4f3e2d1c0b9a
    main.tf
    README.md
    LICENSE
    vars.tf
  /3a9850a2665812933f52423193494b923985b428234857342938472394872349
    main.tf
    vars.tf
```

Each metadata file contains a single Module message which stores the content address of the corresponding module content directory, external source information (for external modules only), and various references to providers and other modules (as proto file content addresses). A reference is included for every direct submodule, direct supermodule, and direct provider usage, but references are not included for transitive supermodules and submodules as they can be resolved transitively. Submodule references include both the content address of the referenced module and the name the module knows the submodule by (for reconstructing the original directory tree exactly). Supermodule references do not include the name, as that information can be retrieved from the equivalent submodule reference in the supermodule's metadata file. The proto is approximately:

```
message Module {
  string content_id = 0;
  ExternalSourceInformation external_source_information = 1;
  repeated SubmoduleReference submodule = 2;
  repeated string supermodule = 3;
  repeated string required_provider = 4;

  message ExternalSourceInformation {
    string hostname = 0;
    string namespace = 1;
    string name = 2;
    string version = 3;
  }

  message SubmoduleReference {
    string hash = 0;
    string directory_name = 1;
  }
}
```

Important caveat: External source information is only present for the top-level module of externally sourced modules, meaning their submodules (i.e. subdirectories) lack any external source information, and transitively linked to their source using the reference system. This avoids unnecessary duplication of metadata and allows deduplication when identical submodules are used across multiple modules.

Note: During the compilation process, module/provider references within module source files (.tf, .tofu, .tf.json, .tofu.json) are updated to point at content address. While they must be transformed again at

runtime, this intermediate state minimises the coupling between the archive format and native Tofu, which aids long-term maintainability and has other subtle benefits for the overall architecture. This detail relates more to the compilation process itself, but it does characterise the archive format. For example, a source file containing `required_providers { aws = { source = "hashicorp/aws" version = "~> 5.20" } }`, would be updated to the form `required_providers { aws = { source = "9c03b11f9b7c85e28a9b3d3b7f191a6c4b2de046559385b2e31e7f193c7d0d69" } }` during compilation.

Providers Directory

The providers directory contains the providers of a Tofu program. It contains a proto file for each provider, and when present, a subdirectory for the provider content. The subdirectories always contain a binary for each of the supported platforms, each named $os_$arch (no file extension), and they never contain nested directories. Below is an example of a providers directory containing three modules, only two of which have content.

/providers

717d8822bb04b2706e41b8cd25f28d4cf041789f9ca69630fb93a0955092d18d.pb

609f901d9e144657e523e6b7f516a1f8d0887ecc55d26e463489cfac9d659ae2.pb

1f2e3d4c5b6a79889f0e1d2c3b4a567890a1b2c3d4e5f6a7b8c9d0e1f2a3b4c5.pb

/717d8822bb04b2706e41b8cd25f28d4cf041789f9ca69630fb93a0955092d18d

linux_amd64

linux_arm

linux_arm64

darwin_amd64

darwin_arm64

windows_amd64

/609f901d9e144657e523e6b7f516a1f8d0887ecc55d26e463489cfac9d659ae2

linux_amd64

linux_arm

linux_arm64

darwin_amd64

darwin_arm64

windows_amd64

Each metadata file contains a single Provider message, which stores external source information (for external providers only) and references to the modules which use the provider in their `required_providers` blocks. References are not included for modules which transitively use the provider as they can be resolved transitively at runtime. The proto is approximately:

```
```

```
message Provider {
  string content_hash = 0;
  SourceInformation source_information = 1;
  string module_hash = 2;

  message SourceInformation {
    string hostname = 0;
    string namespace = 1;
    string name = 2;
    string version = 3;
  }
}
```

Manifest

The manifest file contains a single Manifest message. It records details about the archive itself and details about the Tofu program that cannot be captured in the individual module/provider metadata files. The proto is approximately:

```
message Manifest {
  Archive archive = 0;
  Program program = 1;
}
```

The Metadata message contains information about the archive itself, specifically the version of the archive spec that was used when creating the archive. This provides a degree of future proofing if the spec needs to change to support native Tofu changes. It begins at zero and does not use semver because it's unlike to change frequently. The proto is approximately:

```
message Archive {
    int version = 0;
  }
```

The Program message contains information about the overall Tofu program, specifically the root module and references to external modules/providers. The root field identifies the root module and

inherently prevents multiple root modules from existing, and the external module/provider references allow performant discovery of all external elements (avoids iterating over all elements). The proto is approximately:

```
message Program {
  string root_module_hash = 0;
  repeated string external_module_hash = 1;
  repeated string external_provider_hash = 2;
}
```

Note: In previous design iterations, equivalent XML and JSON manifests were included for human readability, but this was not pursuit to avoid the possibility of manifest divergence (and archive corruption). As protobuf is not human readable, a function was added to the TofuPress program to export the manifest to XML and JSON.

Hashing

Given a directory containing a module/provider, the content address is calculated by hashing the directory name with all its contents. The hash is calculated by transforming the directory into a single contiguous byte stream and calculating the SHA256 digest of the entire stream. The stream begins with the name of the directory, followed by each item in the directory sorted alphabetically, where each item is the full name of the file followed by its contents. Directory/file names are encoded as UTF-8 strings before hashing for platform independence, and file contents they processed without modification. Each hash is encoded in the standard 64-character lowercase hexadecimal representation.

For example, given a directory (foo) with two files, bar.txt and baz.gcl, the byte stream is effectively “foo” + “bar.txt” + <contents of bar.txt> + “baz.gcl” + <contents of baz.gcl>, and the final hash is of the form f4a5c3b2e1d0f9a8b7c6d5e4f3a2b1c0d9e8f7a6b5c4d3e2f1a0b9c8d7e6f5a4. This hashing algorithm assumes modules/provider content directories do not have subdirectories, which is a general condition for the archive structure.

Critical detail: Content directories names and metadata file names both use the same hash, meaning metadata files do not use their own hash, but rather the hash of the directory they correspond to. This details breaks from the traditional definition of content addressable storage but is critical to the effective function of the archive. It allows archives to be merged and reduced based exclusively on the contents of the modules/providers, it allows metadata files to exist without content directories, and it ensures cross-cutting details such as references do not affect the address space. If this were not the case, keeping a separate manifest to associate content with metadata would be necessary, which would negatively affect time and space performance.

Performance

Performance can be understood in terms of four core operations:

- Creation, which is instantiating a new archive.

- Merging, which is turning multiple archives into one.
- Reduction, which is removing content from an archive.
- Export, which is restoring the original directory tree structure.

These operations can be decomposed into:

1. Instantiating the archive structures (creation).
2. Retrieving modules/providers from content addressable storage (merging/reduction/export).
3. Inserting modules/providers into content addressable storage (creation/merging).
4. Removing modules/providers from content addressable storage (reduction).
5. Reconstituting the original directory tree structure (export).

These operations are implemented by the various programs and in practice their runtime performance depends on their implementation, but the archive itself imposes the following hard limits:

- Instantiation involves creating empty directories and an empty manifest, which is a constant time operation due to the lack of parameterisation.
- Retrieval from content addressable storage involves reading the contents of the file/directory that corresponds to the hash of the contents, which is a constant time operation with respect to the number of modules/providers in the archive, and linear time with respect to the number of items in the module/provider.
- Inserting into constant addressable storage involves calculating the hash of the module/provider, creating/populating a directory with its contents, retrieving the proto file for each referenced module, and adding new references. This is constant time with respect to the number of modules/providers in the archive, linear time with respect to the number of items in the module/provider to add, and linear with respect to the number of references to update.
- Removal is the reverse process of insertion and the performance is identical inperformance.

In practice, removal operations are not singular, meaning removing one module/provider also means removing any which are now transitively disconnected as well. Practical removal therefore requires traversal of the reference graph and is a linear operation with respect to the total number of transitively connected modules/providers (since each must be visited to check whether it too can be removed) and constant with respect to the total number of modules/providers in the archive. Furthermore, practical removal often involves removing all external modules/providers, which can be performed in linear time with respect to the number of module/providers to be removed, due to the presence of the external modules/providers lists in the root manifets. Without these references it would otherwise be a linear operation with respect to the size of the archive (due to the need to iterate over each module/provider and check its source information).

These performance limits are made possibly by the combined use of content addressable storage and the reference system. The former allows content retrieval/insertion/removal to scale independent of the total number of modules/providers in the archive, and is limited only by the performance of the hashing operation itself and the need to update references. By keeping both forward and backward references in the metadata files, referenced content can be discovered in constant time (for a module/provider known by content address), meaning the performance of reference updates is limited

only by the total number of references to update, not the size of the archive. These properties are critical for scale at build and runtime.

Note: This performance evaluation assumes the archive has either been fully loaded into memory or unzipped to disk, but if neither are true, then a load/unzip operation must occur first, which is a linear time operation with respect to the number of modules/providers in the archive. This can sometimes be avoided in practice, but not always, and puts a hard limit on theoretical performance. Future work may examine a different file structure which permits random access without preloading.

Emergent Properties

Any given archive has the following properties:

- Correctness, whether it complies with the format described in this document.
- Completeness, whether it contains all transitive dependencies for all included modules.
- Runnability, whether it contains a main module.
- Minimalism, whether it contains only the files it requires for execution.

An archive is correct only when:

- The top-level modules directory exists (may be empty).
- The top-level providers directory exists (may be empty).
- The manifest exists in the top-level and is a Google Protobuf (version 3).
- The top-level modules directory does not contain any subdirectories beyond the first level of subdirectories for each module.
- The top-level providers directory does not contain any subdirectories beyond the first level of subdirectories for each module.
- Each module/provider subdirectory is named as the hash of its contents according to the hash algorithm.
- Every provider subdirectory contains a binary for every supported platform.
- The manifest can be parsed into the Manifest protobuf message without error.
- No extraneous files exist in the archive.
- All module source files (.tofu, .tf, .tofu.json, .tf.json) use content addressing to reference providers and other modules (and never native Tofu addressing).
- The root module has no supermodule references in its metadata file.

The presence of extraneous files and directories is always incorrect to avoid supporting arbitrary files during merge/reduce operations, as these could easily cause conflicts or be handled incorrectly. Any file in the top-level directory which is not the manifest is extraneous. Any file in the modules directory or content directory which is not a content addressed proto file or a child of a content addresed directory is extraneous. Any file in a provider content directory which is not one of the expected binaries is extraneous. Any nested directory within a module/provider content directory is extraneous, as are any files within them. In essence, only the core archive files/directories and the flat list of content items may exist, and all other files are extraneous (and therefore incorrect).

An archive is complete iff it contains the entire transitive closure of modules and providers for its contents (i.e. every module has every other module it requires and every provider it uses). This requires both content and metadata for references modules (i.e. it is insufficient to have only metadata files for referenced modules). An archive which is not complete is necessarily incomplete, but incomplete modules are not necessarily incorrect, and are in fact useful for build-time for performance optimisation.

An archive is runnable if the manifest specifies a root module, and a module which is not runnable is necessarily non-runnable. Note: In both cases, archives themselves are not executable files, and still depend on the Vegan program for runtime execution.

An archive is minimal if it contains only the modules/providers that are transitively required by the root module. An archive which is not minimal is necessarily overloaded, and a module which is non-runnable is only minimal if empty.

TofuPress

TofuPress is a compiler for Tofu. It produces FirmTofu archives from module/provider sources and checks their contents for correctness. It was written in Java for cross-platform support and to avoid coupling it with Bazel (thereby allowing standalone use). It provides four core operations:

1. Compilation, which creates archives from module/provider sources.
2. Merging, which is turning multiple archives into one equivalent archive.
3. Reduction, which is removing modules/providers from an existing archive.

FirmTofu works with files on the local system and does not perform fetching of remote dependencies, as that function is delegated to Soybean. As part of its function as a compiler, it performs For example, when compiling

Note: Archives are treated as immutable, meaning operations which appear to modify an archive actually create new archives and leave the existing archive unchanged. This avoids unintentional data-loss and is simplifies implementation.

Compilation

Compilation creates a FirmTofu archive from native Tofu sources and existing archives. It accepts the following as inputs simultaneously:

1. A root module.
2. A list of non-root modules.
3. A list of providers.
4. A list of other archives.

The compiled artefact is effectively the combined set of all modules/providers from all inputs, but since there can only be one root module in a program, compilation will fail if there are multiple root modules in the combined set. The user is responsible for ensuring that only one root module is supplied.

The program expects all inputs to be provided as local file paths and it does not perform any remote resolution of dependencies (that is handled by Soybean). Source information for externally resolved

dependencies must be passed into the compiler with each module so validation can link them together and verify program integrity. If a module is referenced in another but is not present in the combined set (either due to being elided or because source information is missing) then compilation will fail. Source information can be supplied using textproto files, protobuf binary wire format files, and JSON, all based on the module/provider source messages defined in FirmTofu section. Note: It's valid to pass in module/provider source information without contents, it will simply create a metadata entry in content addressable space without associated content. This is useful when creating library archives that must reference but not contain their external modules/provides.

When processing the root module, it can be used as a validation root or a runtime root, as determined by the no-package-root flag. When the root is treated as a validation root, it is temporarily compiled into the archive, used for validation, then removed. When the root is treated as a runtime root, it remains packaged into the archive and is never removed. This allows validation of library archives which do not include a runtime root, and is necessary because Tofu modules cannot be validated without a root. This is a hard limitation of how the Tofu language works. Internally validation uses the Vegan runtime to unpack the archive before running the Tofu CLI validation function on the result. This ensures the program is both statically valid, passes compilation, and is unpacked correctly. Note: Code is never run as part of this process as that could inadvertently affect production.

When resolving module paths, recursive directories are included by default, and a module will be created in the archive for each directory. This can be disabled by passing in a no-recursion flag. Recursion is the default because Tofu programs outside Bazel are usually structured as deeply nested directory trees, and it would be inconvenient to enumerate every directory.

When resolving provider paths, compilation expects one of three formats in the directory structure, and tries each until one succeeds. First it tries the native Tofu packed structure, then it tries the native Tofu unpacked structure, then it tries a flat list structure (effectively the same structure that FirmTofu uses internally). If the provider directory cannot be parsed into any of these formats, compilation fails with an invalid directory structure argument.

The actual compilation process involves two major steps. First the validation archive is constructed, then the runtime archive is constructed (by removing the validation root if necessary). The first stage involves allocating a temporary working directory, transforming the code into a FirmTofu structure, then zipping the directory into an actual FirmTofu file. The second stage is effectively a reduction operation and is covered below in the reduction section.

The first stage is characterised by the transformations it must apply to the source files. After the process every module and every provider will exist in content addressable storage with a metadata file

The first stage involves the following operations:

1. For each provider, format its contents using the FirmTofu structure (i.e. flat list of binaries named $os_$arch) and insert it into content addressable storage. Next create a metadata file with its source information and insert it into content addressable storeage. At this point it will contain only source information, no references.
2. For each module, insert its contents into content addressable storage, and create a metadata file with the known information. At this point it will contain some but not all of the references, as some are yet to be discovered. As each module is processed, update the references for

A few other design notes:

- The compilation operation can be performed on-disk or in-memory, and the program accepts a configuration value to set this. The former is necessary for archives which are large enough to overload memory and must be progressively constructed on disk before being zipped. This divergence affects the underlying implementation but does not affect general compilation and there result is the same form both paths.
- If two inputs contain the same module/provider (by content address), the compiler can check they have the exact same content. In theory they always should due to content addressing, but in practice it might be necessary to check. For performance this is disable by default and can be enabled by passing the check-hashes-on-merge or check-hashes flags (the former checks all contents).

Merging

Merging archives is the process of creating one archive that represents the combined set of multiple other archives. It's equivalent to a compilation operation with archive inputs only (i.e. no root module, no modules, and no providers). An error is raised during merging if:

1. The archives have different versions.
2. The archives have multiple root modules (collectively).

This operation is effectively a convenience wrapper to simplify usage.

Reduction

Reducing archives is the process of removing content from an archive to create a simpler (i.e. reduced) version.

Supplementary

1. Verify Content
2. Validate Content

Operation 1 scans through the content in an archive and confirms the hash for each entry is correct. It requires recomputing the hash for every entry, which could be inconvenient depending on the size of the archive and the resources of the host, so it is provided as an optional operation, and not an integral part of the core operations.

Operation 2 recompiles the contents of an archive to verify it's still compiled correctly. It works by decompiling the program using the embedded vegan program then compiling them again. It's offered as a supplemental operation to check archives which have been manually modified outside TofuPress or may have become corrupted.

Soybean

TODO jack update this so they output FirmTofu archives for simplicity

Soybean handles downloading Tofu modules and providers at build time so they can be used in the Bazel build graph. In essence, it's the bridge between Bazel's model of build-time external dependencies and Tofu's model of runtime dependencies. It was written in Java to support cross-platform build/execution, functions as a standalone executable, and can be used outside Bazel.

Soybean exposes the following functions:

1. Download module.
    1. Inputs:
        1. Module (spec). The fully-qualified address of the module to download.
        2. Output (path). The location to write the downloaded files to (directory)
    2. Outputs: A directory containing the downloaded module sources, including transitive module dependencies, but not provider binaries.
2. Download provider.
    1. Inputs:
        1. Provider (spec): The fully-qualified address of the provider to download.
        2. Output (path). The location to write the downloaded provider files to.
    2. Outputs: A directory containing the downloaded provider sources for all architectures/OSes.

Internally soybean uses the Tofu CLI to perform the download. It generates a basic tofu file that uses the required provider/module, then calls the CLI to perform the download. After it has downloaded the dependencies, it uses TofuPress to create a FirmTofu and outputs it.

TODO more implementation details

Vegan

The Vegan program operates on the archive supplied as an argument, but previous design iterations focused on baking the archive into the Vegan program through various means, such that running a given instance of Vegan meant operating on a hardcoded configuration. While aspects of this approach are retained in the output of the tofu_binary rule (discussed in more detail in the build rules implementation section), the Vegan program was made independent from any one particular archive to allow isolated compilation and testing, and to unlock the option to release archive execution functionality to non-Bazel users.

TODO implementation details

Supplementary

The following supplementary operations are provided:

1. Export the manifest to a human readable format.
2. Lookup a module from source information.
3. Lookup a module from content address.
4. Lookup a provider from source information.
5. Lookup a provider from content address.

6. Lookup the submodules of a module.
7. Lookup the supermodules of a module.
8. Lookup the providers of a module.
9. Lookup the modules of a provider.

Operation 1 accepts a flag specifying JSON or XML and exports the manifest in that format. It exists because reading the manifest is useful during debugging but the protobuf-based manifest is not human readable.

Operation 2 accepts the source information of a module and returns its content address, and operation 3 does the opposite. Operations 4 and 5 are equivalent operations for providers. Operations 6, 7 and 8 accept the content address of a module and return the content addresses of its submodules, supermodules, and providers (respectively). Operation 9 accepts the content address of a provider and returns the content addresses of the providers which use it. These operations exist to allow graph traversal which can be useful during debugging.

BakedTofu

BakedTofu executables are Java JARs which can be run to provision infrastructure. Runtime Execution is the process by which a BakedTofu archive is used to provision infrastructure. It begins when a machine (either a developer workstation or CD machine) runs a BakedTofu executable with the JRE, and completes when the JRE exits. The process is as follows:

1. The JRE begins executing the main program.
2. The main program copies the BakedTofu archive and the vegan runtime to a temporary directory.
3. The main program invokes the

BakedTofu archives are self-contained binaries that contain a complete FirmTofu archive, the Vegan runtime, and a basic main program to link them together. , so they can be executed at runtime to provision the infrastructure described by the FirmTofu archive. Runtime execution follows this process:

TODO

Library Rule

Create compilable tofu libraries that can be composed and validated automatically, with support for both root modules and non-root modules.

```
tofu_library(
  name,
  srcs,
  deps,
  is_root,
  validation_root,
) -> FirmTofu
```

Library rule prevents files from subdirectories because that does not work with the module structure of tofu. Easy runtime check in implementation.

When a library targets is built, the output is a lightweight archive for performance

Library will only allow .tofu .tf .tofu.json and .tf.json files, and everything else can do in data. In practice both sets will be merged together into the contents of the library, but this separation is consistent with the bazel approach to sources and runtime data being declared separately.

All rules require deps to be non-file targets for consistency with broader bazel conventions.

Note: The Tofu Library rule builds a reusable component for use as a dependency in other targets, and the underlying data structures are optimised for performance by minimising the artefact contents and relying on providers for the . The Tofu Export rule builds a reusable component for deployment to external package managers, and the underlying data strutures

Non-Root Module Validation

Problem: Non-root modules cannot be validated in isolation because they need a root module to define their provider configurations. If one were validated in isolation it would fail. There are no good options for this, because it requires:

Solution: Allow bazel rule users to provide a validation target which is effectively a template for a root module which performs all the necessary configuration. It would just be a flat list of provider configurations. This would require one point in the codebase to consolidate configurations for all known usages though which might get hard to manage. I wonder if it can be managed more granularly. It would probably need a new rule.

```
tofu_validation(
  name=”child_module_validation”,
  srcs = [tofu files],
  deps=[providers/modules],
)

tofu_library(
  name = “child_module”,
  deps = [providers/modules],
  validation_root = “:validation”,
  is_root_module = False,
)
```

Implementation wise the validation target will have a placeholder for a child module and will not be validated itself. When it’s passed to the validation line in tofu_library, the new module gets patched into the placeholder, and the patched module gets used as the root.

To reduce the burden (adding a validation target to every lib) it could be possible to define this as a toolchain and apply it across the repository automatically. If the validation_root is set, it would override the toolchain, and if is_root is True, the toolchain is not used. Would need a toolchain per Bazel module though, which could be hard.

Note: It would be an error to supply a validation argument and have is_root_module True. Rules will check for this during build.

Export Rule

Export modules for publication.

```
tofu_export(
  name,
  deps,
  exclude_external_modules,
) -> FirmTofu with external deps potentially removed
```

Binary Rule

Create executable tofu programs that can be run to provision infrastructure.

```
tofu_binary(
  name,
  deps,
) -> BakedTofu
```

Test Rule

```
tofu_test(
  name,
  deps,
  srcs,
  testfile,
)
```

Uses the Tofu CLI test command internally.

Since the test interacts with processes outside Bazel (infrastructure) it is non-hermetic. Bazel must be made aware of this with the `no-cache` tag. Furthermore, the `requires-network` must be included to allow the test access to the network (for connecting to remote infra). Together these tags configure Bazel to allow network access and re-run the test whenever requested (as opposed to returning the cached result).

Toolchains

Toolchains are a standard part of Bazel and are relatively straight forward. Each one wraps a tool and exposes it to build rules in a way that hides platform-selection details. The only point of nuance is where they source their tools. The Tofu CLI is sourced from the external tool repository created by the Tofu CLI repository rule, and the custom program tools are sources from the build targets which compile them on-demand.

Module Extension

The module extension has two responsibilities:

1. Downloading the Tofu CLI and registering the Tofu CLI toolchain.
2. Downloading the module and provider dependencies into external repositories.

Operation 1 is straightforward. The module extension provides a tag to specify which Tofu CLI to use as an HTTP address pointing to a download location. It is optional and may be elided to specify the latest version from GitHub. It's an error to set it repeatedly though and doing so will cause the build to fail. In the module extension itself, the Tofu CLI Repository Rule is used to perform the download, and a toolchain is registered using the standard toolchain API. Overall this makes the Tofu CLI available to build rules without dependence on the host system.

Operation 2 is more complex because it involves transitive dependency resolution and version conflict resolution. While providers themselves are pre-compiled binaries and thus have no transitive dependencies, modules may depend on both modules and providers, so given a list of modules and providers, the transitive closure may include other modules and other providers. Furthermore, as modules are not globally coordinated in any way, multiple versions of the same module may exist in the transitive closure, and a strategy is required for handling them.

In order to be intentional about versioning and reduce the possibility of dependency hell, the ruleset follows an approach similar to Bzlmod versioning, whereby it resolves a single version for each dependency by default, but allows ruleset users to specify version overrides as required. Reusing that approach for Tofu module/provider resolution ensures a single-version strategy is the norm without locking ruleset users into dependency hell when versions require fine tuning. Implementation wise this means the module extension automatically upgrades every module to use the latest version in the dependency graph, but allows ruleset users to specify specific dependencies to not upgrade.

For example, consider dependency A which has versions 1.1. and 1.2 in the resolved transitive closure. Most modules which depend on A may work with version 1.2, but one specific dependency, dependency B, might only work with version 1.1. In that case, the user must specify that dependency B, but only dependency B, can use version 1.1 of dependency A, while all other versions continue to use version 1.2. This ensures the majority of the graph is using the latest version and positions overrides as a workaround not an ideal solution.

An important details is the handling of transitive dependencies of the overridden module. While the override process allows a specific module to retain a dependency on a specific version of another module/provider, the retained module may still depend on other modules, and those versions will be

upgraded. In order to specify a version override for the entire transitive closure of a module, a version override must be set for every transitive dependency. This prevents the top-level configuration from becoming cumbersome and keeps version overrides focused on the specific module in question.

For operation 2, the module extension provides four tags:

1. A tag to specify first order modules dependencies.
2. A tag to specify first order provider dependencies.
3. A tag to specify modules version overrides.
4. A tag to specify provider version overrides.

All tags are optional and can be elided to specify no dependencies/overrides. Tags 1 and 2 accept a dictionary where the keys are HTTP addresses pointing to download locations and the values are version strings. Tags 3 and 4 accept two parameters, the first to specify the target for the version override and the second to specify the dependency to retain (i.e. which has a dependency to keep, and which dependency to keep).

The module extension resolves the transitive closure of all first order dependencies, applies upgrading with respect to version overrides, then creates a flat list of all modules and providers. The lists are then passed to the repository rules so they can be downloaded and exposed as targets. In order to keep the transitive dependencies isolated from the first order dependencies, first order dependencies and transitive dependencies are stored in different repositories, and the transitive dependencies repository is only visible to the first order repository. This prevents ruleset users from accidentally depending on the targets for transitive dependencies in their build targets.

Since version resolution happens in the module extension it will be evaluated as soon as any target in a build execution depends on a single external module. This is not ideal for performance, but cannot be avoided, as the entire transitive closure is required for version resolution.

The alternative of allowing multiple versions by default was considered, but it does not align with the Bazel philosophy or general engineering best practices, as it creates a significant opportunity for dependency hell and does not scale to large repositories. For these reasons the single-version approach was selected instead.

The module extension and repository rules presently depend on the Soybean program for downloading external dependencies, but two alternatives were considered. First, the Tofu CLI could have been invoked directly in the extension and rules, and second, the CTX.download function could have been used to perform the download using Bazel's built in logic. Both eliminate the need for the Soybean program entirely, and while that may reduce typing, it prevents rigorous unit testing of the download functionality, and in the case of option two, would require reimplementing complex ecosystem-specific authentication/fetching logic in Starlark. For maintainability, Soybean was used. In summary, the module extension and repository rules use Soybean internally to avoid reinventing the wheel.