‘Ten’ DSL

Pure Functional Build System & Unified Declarative Syntax

Enzo Joly, 22055453

Module Title: UFCFFF-30-3 | Software Development Project

Word Count: 7,400

Abstract

A proposal and implementation for Ten, using Haskell’s strong typing to create a pure functional and hermetic build system with isolated effects for deterministic software deployment. Both transactional and atomic, it aims to solve deployment and make it portable for any version of any software on any machine.

The concept is based on Nix, and implements a garbage-collected content store with derivation-based dependency resolution and hash verification of inputs.

Haskell, with its lazy evaluation, lends itself to a pure functional model with strict composition of side-effects for a declarative DSL (Domain Specific Language), specifically for building and deploying software. One key aim of the Ten project is to facilitate streamlined prototyping of build system architecture. Haskell is also well-suited for concurrent or parallel build processes, allowing for performant prototypes.

---

https://github.com/enzojoly/ten
https://github.com/enzojoly/ten-mvp

Contents

1 Introduction

1.1.0 Problem Statement

Software deployment has historically been plagued by fundamental challenges that undermine reliability, reproducibility, and maintainability:

1. **The Dependency Problem:**
   Software depends on library versions, which themselves have dependencies. Conflicting requirements create a "dependency hell" where satisfying all criteria becomes impossible.
2. **The Mutation Problem:**
   Traditional package managers mutate shared system state in a non-atomic and potentially destructive manner, leading to unpredictable system behaviour after updates or installations.
3. **The Reproducibility Problem:**
   Recreating identical execution environments across machines or over time remains practically impossible with conventional tools. "Works on my machine" persists as a development truism.
4. **The Composition Problem:**
   Combining software components from different sources creates brittle systems where modifying one component can unpredictably break others through hidden interactions.
5. **The Isolation Problem:**
   Maintaining multiple versions of the same software for different applications introduces complex configuration management challenges that existing isolation techniques (containers, VMs) only partially address.

To sum up:

- **Dependency Hell:** Applications requiring conflicting versions of the same libraries
- **Global State:** Traditional package managers modifying system-wide state
- **Poor Reproducibility:** Builds producing different results on different machines
- **System Inconsistency:** Partial updates leading to broken states
- **Limited Isolation:** Inability to run multiple versions of software simultaneously

1.2.0 The Anatomy of an Existing Solution: Nix Flakes

Nix solves software deployment through five fundamental principles that form its mathematical foundation:

$$\mathcal{P}_1 : \text{Content-addressable store} \Rightarrow \text{paths derived from content hashes} \tag{1.1}$$

$$\mathcal{P}_2 : \text{Pure functional evaluation} \Rightarrow f(\textit{inputs}) = \textit{outputs} \text{ deterministically} \tag{1.2}$$

$$\mathcal{P}_3 : \text{Lazy evaluation} \Rightarrow \text{builds only what is needed} \tag{1.3}$$

$$\mathcal{P}_4 : \text{Immutable store} \Rightarrow \text{once built, never modified} \tag{1.4}$$

$$\mathcal{P}_5 : \text{Isolated build environments} \Rightarrow \text{no leakage of dependencies} \tag{1.5}$$

Flakes extend this foundation by addressing critical deficiencies in traditional Nix usage. They formalise an approach that makes Nix packages more accessible, composable, and maintainable:

1. **Reproducibility:**
   Flakes enforce *hermetic evaluation*, meaning the build depends only on explicitly declared inputs. This eliminates the implicit environment capture that plagued earlier Nix usage.
2. **Composability:**
   Flakes establish a formal method for depending on other flakes, creating a directed acyclic graph of dependencies.
3. **Extensibility:**
   The `flake.nix` schema imposes minimal constraints beyond basic structure.
4. **Usability:**
   The `nix run` command leverages the standardised interface to execute flakes without extension to the default configuration. "Just works"
5. **Discoverability:**
   `nix flake show` provides structural introspection without code analysis. Everything you need to know can be gleaned from the command line without documentation.

Nix treats software deployment as a pure function with explicitly declared inputs, content-addressable storage, and immutable build artifacts. This provides a principled solution to problems that conventional package managers, by their stateful design, cannot fundamentally resolve.

1.3.0 Lazy Hermetic Build Evaluation with Functional Purity

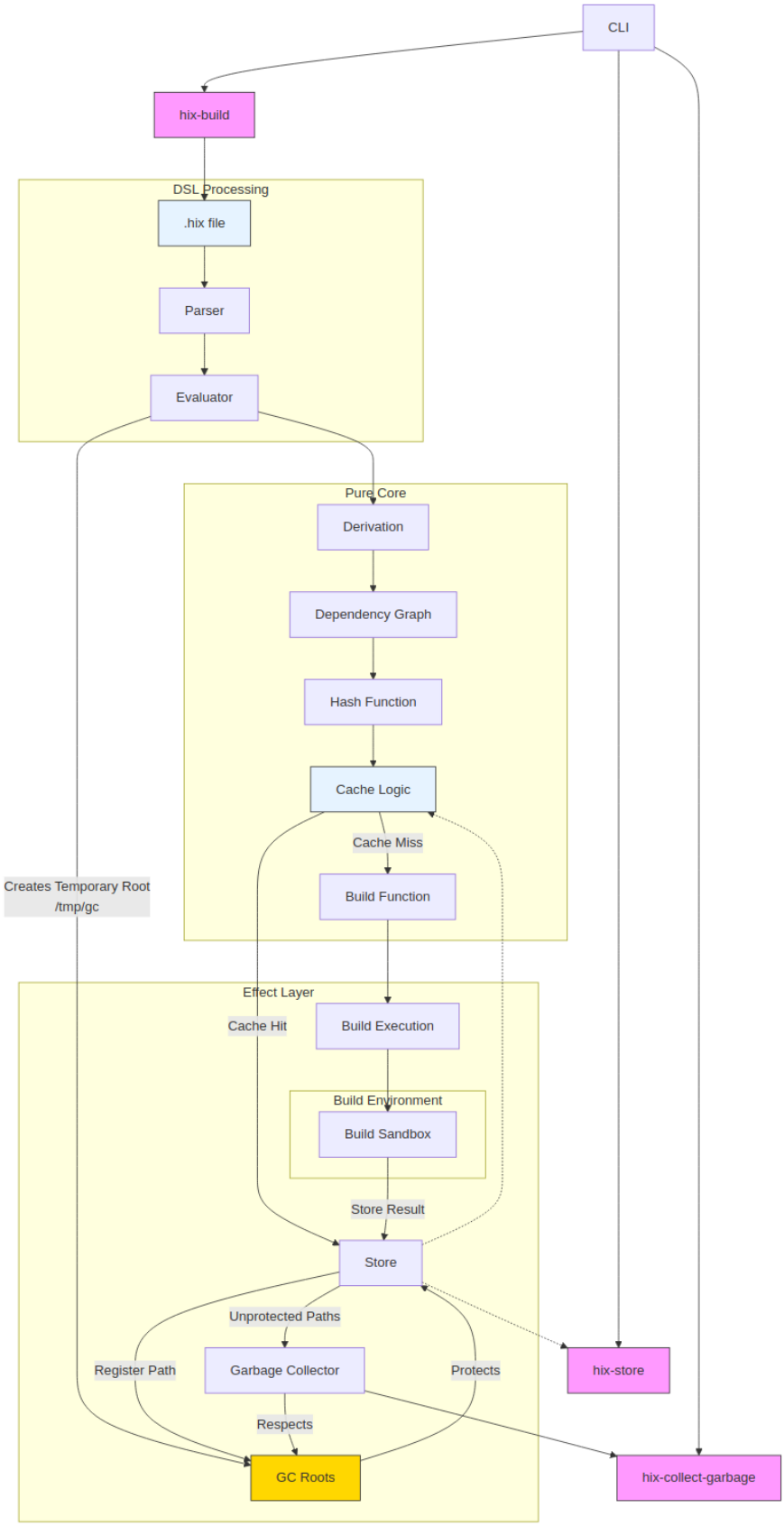
The initial proposal of Ten revolves around a stripped-back iteration of the flake structure pioneered by Nix. Though it was never a requirement that Ten be an original or novel implementation, it is the case that Ten was devised with a propsective approach in mind, and to be a prototyping and experimentation framework that might be used as a research tool, as opposed to one fit for use in production.

Ten's key risks include not finishing within the deadline, not solving for the necessary functional and non-functional requirements, and finally a low quality standard generally across the implementation in the end product. We mitigate this in part through the use of Haskell, which will keep the codebase concise and reusable, easily maintainable, and adjustable. It has and will continue to be imperative to the success of the project that much UML be devised and experimented with. We can not rely on traditional object-oriented methods for system modelling.

Such a project is liable and susceptible to suffer from scope creep. An attempt is made in the proposal to break Nix into its most fundamental and required functional promises. A CLI with three commands (build, store, garbage-collection), and the DSL, Pure Core, and Effect Layer for support of these three commands. That's it!

N.B. Initially the project was termed Hix (Haskell Nix) but has since undergone a name change.

1.4.0 Project Proposal: DSL Tri-Layer Facade Model

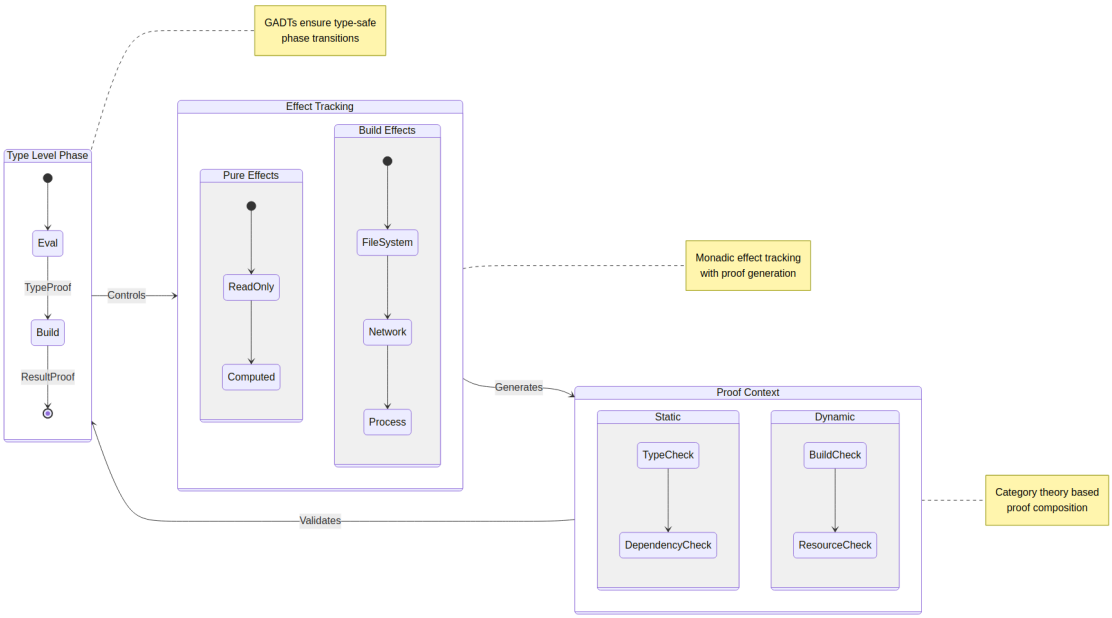


2 Methodology

2.1.0 Formal Verification & Category Theory

At the outset of the Ten project, we begin with nothing but definitions for the mathematical proofs that transformations preserve essential build properties. With these proofs, revisiting them if necessary, we model our first functional requirements for the Ten Build System. These loosely resemble the inverse of the aforementioned problem statements from our introduction.

"Category theory is a way for seeing the world, It should be viewed not so much as one particular area of mathematics among many others but more as a method for unifying much of mathematics." *(Garrity 2021, p. 345)*



As we start to flesh out the core function for Ten, a build system which by definition serves to provision specific morphisms (mutation, changing inputs and building them into outputs) upon existing structure, this implementation naturally progressed from proof to category-theoretic model.

- **Objects**: Software artifacts (source files, intermediate outputs, final binaries)
- **Morphisms**: Build transformations between artifacts
- **Composition**: Sequential application of build steps

- **Identity**: The no-op build step that leaves an artifact unchanged

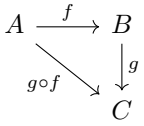


Figure 2.1: Category composition in a build process, general rules that transcend context

2.2.0 Data Synthesis

Given the niche subject matter, and a poor state of affairs with respect to Nix's documentation, a large amount of high-specificity research was undertaken with LLMs in order to explore context-specific perspectives relating to functional build systems, as well as comparing and contrasting existing alternatives. These so-called 'AI' models are an additional risk-factor element if misused or over-relied upon given their inadequacy at forming coherent big picture ideas across any substantial scope and/or scale. Codebases more than a few thousand lines up until only very recently were virtually an impossibility for chatbot inference to grasp, though the sheer magnitude of output from NLP tooling, and the hyper-specific answers that can be gleaned when narrowing the context window significantly, and by way of comparing between models, making them second guess themselves - was without doubt an asset to this project in its research, development, and testing phases respectively.

UML diagrams were sometimes generated with Anthropic's Claude Sonnet 3.5+ and later Claude Sonnet 3.7 Extended Thinking into Mermaid and PlantUML syntax. They were then compiled into svg and converted for high quality. Often the diagrams were unused though operational semantics were better carried between LLM inference using these schematics. A convergence towards higher quality was observable with manual alteration to build plans and diagrams, allowing for minute details in context and purposeful prompting to direct these models to avoid their common pitfalls and meticulously crafting shaped vessels for outputs, ultimately ensuring final products were culturally entrenched in human oversight for the best results.

2.3.0 Technology Selection, Versioning & Document Control

Code Language

We will use `ghcup` for our Haskell toolchain to install, set, and manage versioning for:

- Glasgow Haskell Compiler 9.8.2 (GHC)
- Cabal 3.1.2
- Haskell Language Server 2.9.1 (HLS)

UML

We will inform development heavily with diagrammatic representation:

- AstahUML
- Mermaid
- PlantUML

Documentation/Version Control

We will save progress to our work regularly and transparently, documenting as we go:

- LaTeX
- plaintext (notes taken in NeoVim)
- Haddock (for generated documentation)
- GitHub

With regard to LLMs, we should aim to keep abreast of the current trends, at the time of this writing, Claude Sonnet 3.5+ and 3.7 Extended Thinking seem to be frontrunners as far as the state of the art, though Deepseek R1 has made some waves and I personally prefeer it. I expect to use Gemini 2.5 Pro, and Grok 3's Think and DeeperSearch modes also.

The technology selection worked backwards from a desire to create a Nix-like system in Haskell. This is also my first experience with Haskell, though even paying no heed to an existing motivation to use it, it is nonetheless greatly suited to a build system relying on pure functions dynamically and fares well if not better than alternative language options for this use case. Rust might ultimately be better suited for a long-term project looking to innovate directly on the Nix model, since it is a systems language and can grant developers incredible flexibiblity and utility with its zero-cost abstraction, but especially for this project under time constraints Haskell may still be the superior choice. It allows us to move fast and, as a mathematical language, graph modelling and traversal are a breeze. For formal documentation, Haskell also supports Haddock-style comments that start with {-| or –|: and can be generated automatically.

We first initialise a Cabal project and set our dependencies, ironically looking to avoid Cabal Hell right from the first hurdle (one day we won't have to worry about you anymore!) and here in our `ten.cabal` file is where we specify details about the project, the versioning of dependencies and the module structure that the code adheres to.

This versioning combination is not the only possible pick, but this nets us a great deal of power, and with real-time meta-analysis of our code thanks to HLS, as if we were testing on the fly. An LSP can greatly speed up the pace of development and allows us to patch errors on the fly. I also added this mapping to 'remap.lua' (in my custom lua NeoVim configuration) in order to parse directly between LSP error diagnostics directly and speed up the debugging process:

```
-- Jump to the next diagnostic (e.g., error)
map('n', 'f', vim.diagnostic.goto_next, { desc = "Go to next diagnostic" })

-- Jump to the previous diagnostic
map('n', 't', vim.diagnostic.goto_prev, { desc = "Go to prev diagnostic" })
```

3 Research

3.1.0 Deciphering Project Scope, Following in the Footsteps of Giants, Reading Dolstra's PhD Thesis...

The research undertaken for this project was a totally free and by extension totally intimidating aspect to this work. It was precisely this freedom that made it so daunting, yet so exciting, but also it meant that the clock was ticking all the time with no ideas tabled as to deciding what mission was to be embarked upon. Forgive me if I speak candidly here for the work that went into this chapter on reflection grew to become the most engaging of my unique personal efforts, and perhaps the more demanding of me to actually adapt and think, and the undertaking itself was perhaps the single highest ROI chapter of this whole project in hindsight.

From start to finish - from the moment I had the idea for Haskell Nix, I would suggest that there were three thoughts that plagued my mind, concerns that stood above the rest as most pressing and most significant of all:

- "I have zero Haskell knowledge": I required to learn Haskell
- "I have minimal previous FP experience": Object-Oriented Systems being my only priors
- Whilst learning the above, create a fully functional Nix-like system with no prior build system experience? At least Nix's code is open-source, and the whole thing exists already!

I tackled these concerns with a three-pronged blind assault free-falling, kicking, and screaming into the darkness. For Haskell knowledge I first commenced meta-learning the entire langauge from a very high level, and drawing up a study plan for Haskell: "What practices exist?" I thought. I asked this question first, though I understood the draw of functional programmes and I immediately began hoping it would solve the other problems I was facing, I started on creating a long list from beginner to advanced paradigms, structures, methods, best practices and one-by-one I studied the loose definitions. But I knew that hands on would be the only effective solution. I picked up 'Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming' *(Skinner 2023)* and it worked well to springboard me into the world of Haskell. I watched some interviews with the great Simon Peyton-Jones, and some excellent lectures from the likes of the marvellous Brian Beckman, Philip Wadler, and Bartosz Milewski, and I also made myself very familiar with some of their literature and attempted to retain it as best I could.

N.B. Specifically in the case of Bartosz Milewski I endeavoured to track through the entirety of his "The Dao of Functional Programming" (Milewski 2025), and apply as many concepts as I could to a Nix-like build system. My notes can be found in the main repo under documentation/wiki (dao for build systems)

All in all and some months later, I had my category theory straight, I finally knew what is a monad. And yet I didn't feel a whole lot closer to building a cohesive build system environment. They say

that X.com is the world's town hall, but I would argue you find what you seek, and by my good fortune I stumbled upon one user @HSVSphere hailing from the land of Turkiye. Allow me to step backward in time before I step forward, and say that I owe the inspiration of this project to two people in particular at least by my own rough approximation. One of those people, the naked flame to the gasoline in this instance, was Diego - a Japanese research scientist at KDDI, and self-proclaimed Nix-enthusiast, whom I had the pleasure of speaking with at the KDDI Summit in early September of 2024.

At the time I was as I am now, an avid NeoVim maxi, but had shyed away from Nix - perhaps daunted by the immense learning curve and what I will term "the documentation situation". The two go hand-in-hand however, as Nix can make NeoVim configurations portable (ergo NixVim) and Diego insisted I give it a try. It would be another month or so before I modelled the first prototype for "Hix" (Haskell Nix) which came after abandoning many worse ideas for a software development project focus. But with this initial nudge in the right direction I was turned onto the world of declarative build recipes, and their benefits. It was at this time that I reached out to HSVSphere, one of the prevalent talking heads whom seemed to understand the value in these systems also, and is working on his own, very much different, contribution to the PLT world by the name of "Cab" (you can check it out here). He is looking to make something entirely new, however, where I was looking to understand what is already out there much more intimately and in the aire of the grasp he had of it. I sent him the project proposal model displayed in section 1.4.0 and asked his thoughts.

3.2.0 A Discussion with HSVSphere

As Diego sparked the inspiration as it were, Sphere's equivalent contribution would be in coaxing the idea to its correct destination.

Our conversation was short and sweet but hinged on the fundamental elements that a "slightly better Nix-clone" might demonstrate. Sphere made a few key suggestions:

- *"You shouldn't allow IFD (Import From Derivation), clearly separate Eval and Realise steps"* - this became the Phase type separation system.
- *"Do you know how ReaderT in Haskell works? You should do the same for transitive config management of Thunks"* - Nix has a `config.allowUnfree` setting that users need to set globally to allow the use of packages with non-free licenses. This is set at the global/config level and doesn't transitively apply to specific thunks or package definitions easily. Just riffing here, since I do not have a full grasp of this myself, but I believe were Ten to be fully implemented, and perhaps even now the structure to enable this may exist in the code as it stands, since the build monad (TenM) and its monad transformer stack is fully implemented, though the DSL layer is not.

  Looking at Ten's core implementation, and we cover this later, but it uses ReaderT as part of its fundamental design:

```
newtype TenM (p :: Phase) (t :: PrivilegeTier) a = TenM
    { runTenM :: SPhase p -> SPrivilegeTier t -> ReaderT BuildEnv (
        StateT (BuildState p) (ExceptT BuildError IO)) a }
```

  The TenM monad is built on top of ReaderT BuildEnv, which means:

  – Configuration is explicitly passed through the `BuildEnv` structure
  – Any derivation can access this configuration via the Reader pattern
  – Configurations can be modified locally for specific subtrees of computation

  The relevance of this explicitness is significant:

  – Type Safety: Configuration access is statically checked at compile time

  - Locality of Effects: Configuration changes can be isolated to specific derivations or functions
  - Transparency: Dependencies on configuration are explicit in types, making them part of the API contract

  In practise, this means that rather than having a single global `config.allowUnfree` setting like in Nix, Ten would likely allow you to modify configuration for specific subtrees of your build graph:

```
-- Hypothetical example of how Ten might handle this
buildWithUnfree = do
  -- Get the current environment
  env <- ask

  -- Run a computation with modified environment
  withUnfree <- local (\e -> e { allowUnfree = True }) $ do
    unfreePackage <- derivation { ... }
    return unfreePackage

  -- Outside that scope, original config applies
  regularPackage <- derivation { ... }

  return (withUnfree, regularPackage)
```

- *"One original thing you should do is Recursive Hix. The idea is a FIFO or Socket in the Build Env and write a derivation tree to it. Then kill yourself and let the builder replace your node with the trees root. The biggest challenge for this is setting up the isolated build environment fast."* - since the rabbit hole goes deeper, this was left unresolved with respect to the final frontier. My working solution was in essence a mere rehash of a pull request called RFC#40 on the NixOS Master branch (which Sphere also pointed me to).

  But, that being said, this is how the first inklings of the Monadic build recipe structure for Applicative and Monadic builds came to be (see Section 5.5.0). And the beginnings of my idea for a Unified Syntax. Sphere's notion was to *"track FS events and if nothing had changed in the Build Env (not `$out`) then reuse it"*. I had no time or will spare (not for lack of wanting) to investigate this myself but the idea was definitely worth a mention here. He envisioned, from what I can gather from text, a method to take a CMake file (turn it into a derivation tree?) and turn a single LLVM derivation into a deduplicated, optimised mass of micro-LLVM derivations and, as he put it, *"get incremental compilation for free"*. I may be butchering his meaning here, in fact I surely am, but it is an excellent idea, and was nonethless insightful and at the time I applied what I could, even if such a problem as this one was out of the project's scope when taken to its exhaustive limit. Even when the mechanics of this idea were totally beyond me the end results were well understood and, in my opinion, they are a step forward into the future of the field.

  In Ten, every derivation writes to `$out`. This is the standard environment variable representing the path in the content-addressed store where build outputs should be stored.

  For derivations that participate in Return-Continuation (a multi-stage build approach), there's an additional special path called `returnDerivationPath` which is defined as:

```
returnDerivationPath :: FilePath -> FilePath
returnDerivationPath sandboxDir = sandboxDir </> "return.drv"
```

  This is exposed to the builder as the environment variable `$TEN_RETURN_PATH`, which allows a derivation to output another derivation that will be built next:

```
cat > $TEN_RETURN_PATH << EOF
```

  So, while every derivation writes its main build artefacts to `$out`, derivations that participate in bootstrap pipelines or multi-stage builds can additionally write to `$TEN_RETURN_PATH` to specify the next derivation to be built.

4 Requirements

4.1.0 Functional & Non-Functional Requirements

4.1.1 User Stories

User stories capture the needs and goals of different stakeholders who will interact with the Ten build system. They form the foundation for our requirements and drive the development priorities.

ID, Actor	User Story
US1, Developer	I want reproducible development shells so that my software works consistently across different environments.
US2, Administrator	I want isolated builds so that different versions of the same software don't conflict.
US3, Team Lead	I want declarative package management so that builds are maintainable and verifiable.
US4, DevOps Engineer	I want efficient caching so that rebuilds are fast and resource-efficient.
US5, Security Engineer	I want sandboxed builds to prevent security vulnerabilities during the build process.
US6, Platform Engineer	I want to be able to track dependencies precisely so that I can understand the complete software supply chain.
US7, Release Manager	I want atomic upgrades and rollbacks so that system changes are safe and predictable.
US8, Systems Integrator	I want composition of build artifacts so that I can create complex deployments from simpler components.
US9, Project Maintainer	I want clear error messages so that I can efficiently debug build issues.

4.1.2 ISO/IEC 25010:2023

ISO/IEC 25010:2023 elicits and defines a comprehensive quality model for software and information system requirements products. They allow us to identify product and information system testing objectives as well as quality control and acceptance criteria. They allow us to validate the definition of our requirements thoroughly, and establish and support measures for product quality characteristics.

4.1.3 Functional Requirements

FR1: PURE FUNCTIONAL CORE REQUIREMENTS

A core based on pure functional principles, ensuring deterministic builds and immutable storage.

FR1.1: Referential Transparency (Core)
FR1.2: Immutable Store (StorePath)
FR1.3: Declarative Specifications (DSL)

- The system must prevent side effects from affecting reproducibility, ensuring that the same inputs always produce the same outputs, and that build results are deterministic regardless of outside environment or context
- All build outputs must be immutable, preventing modification of stored artefacts, and creating new paths for new versions. The store must maintain hash-based addressing for all artefacts
- Build recipes must be purely declarative, supporting functional composition of build steps, and enabling reasoning about builds as pure functions

functional correctness, the "capability of a product to provide accurate results when used by intended users" *(ISO 2023)*.

integrity, the "capability of a product to ensure that the state of its system and data are protected from unauthorised modification or deletion either by malicious action or computer error" *(ISO 2023)*.

functional completeness, the "capability of a product to provide a set of functions that covers all the specified tasks and intended users' objectives" *(ISO 2023)*.

Satisfies user stories: **US1**, **US2**, **US7**, **US3**, **US8**

FR2: DSL INTERFACE FOR BUILD SPECIFICATION

A domain-specific language for build specifications that maintains functional purity.

FR2.1: Ten Expression Language (Parser)
FR2.2: Functional Path References (StorePath)

- Must provide a purely functional DSL for build specifications, handling path references in a purely functional manner, and supporting complex dependency specifications
- Must handle path references in a purely functional manner, validating paths at evaluation time, and preventing invalid path references

functional completeness by providing necessary expressive capabilities.

functional correctness by ensuring valid references.

Satisfies user stories: **US1**, **US3**, **US8**

FR3: HOLISTIC EXECUTIVE COHESION OF MODULAR ARCHITECTURE

A principled, layered approach that maintains functional purity.

FR3.1: Pure Functional Builds (Build)
FR3.2: Content-Addressed Storage (Hash)
FR3.3: Immutable Store (StorePath)
FR3.4: Garbage Collection (GC)
FR3.5: Layered Architecture (Monad Stack)
FR3.6: Daemon-Based Operation (Privilege Tier)

- Must define builds as functions that take inputs and produce deterministic outputs, isolate build effects, and track all dependencies explicitly
- Must derive store paths from content hashes, use cryptographic hashing for integrity, and provide content verification
- Must ensure derivations are immutable once built, prevent unauthorised modifications, and maintain content integrity

- Must remove unreferenced items from the store, preserve referenced items, and provide safe collection mechanisms
- Must implement core storage, derivation, and package management layers through a monad transformer stack with clean interfaces between layers
- Must handle privileged operations through a daemon process, enforce privilege separation, and provide secure client-daemon communication

functional correctness through deterministic execution.

integrity through cryptographic verification and preventing content modification.

resource utilisation, the "capability of a product to use no more than the specified amount of resources to perform its function under specified conditions" *(ISO 2023)*.

modularity, the "capability of a product to limit changes to one component from affecting other components" *(ISO 2023)*.

security by enforcing proper privilege separation.

Satisfies user stories: **US1**, **US2**, **US3**, **US4**, **US5**

FR4: DERIVATION MANAGEMENT

A system to manage derivations with comprehensive dependency tracking.
FR4.1: Derivation Representation (Derivation)
FR4.2: Dependency Resolution (Graph)
FR4.3: Build Strategies (Derivation)

- Must define derivations as self-contained specifications, supporting all necessary build parameters and encoding all build dependencies explicitly
- Must resolve all direct and transitive dependencies, prevent circular dependencies, and ensure closure completeness
- Must support different derivation evaluation strategies, handle fixed-output and input-addressed derivations, and support both applicative and monadic build contexts

functional completeness by providing complete build specifications.

functional correctness by ensuring proper dependency resolution.

functional appropriateness, the "capability of a product to provide functions that facilitate the accomplishment of specified tasks and objectives" *(ISO 2023)*.

Satisfies user stories: **US1**, **US3**, **US6**

FR5: STORE MANAGEMENT

Management of the store efficiently and securely with validation mechanisms.
FR5.1: Store Path Validation (StorePath)
FR5.2: Hash Verification (Hash)
FR5.3: Content Deduplication (Store)

- Must validate all store paths against required format, enforce path integrity, and prevent invalid path manipulation
- Must verify content against declared hashes, reject content that fails verification, and use secure hash algorithms
- Must prevent duplicate storage of identical content, use content-addressed storage for deduplication, and optimize storage efficiency

integrity by ensuring valid store paths and through cryptographic verification.

resource utilisation by preventing unnecessary duplication.

Satisfies user stories: **US1**, **US4**, **US5**, **US7**

FR6: ERROR HANDLING & SYSTEM INTEGRATION

Handle errors gracefully and integrate with external systems.

FR6.1: Sandbox Isolation (Sandbox)
FR6.2: Error Propagation (Build)
FR6.3: System Integration (CLI)

- Must isolate builds in secure sandboxes, prevent cross-build interference, and limit resource access during builds
- Must handle errors gracefully throughout the build process, provide meaningful error messages, and maintain system consistency during errors
- Must provide interfaces for integration with external systems, support standard build input/output formats, and enable use in CI/CD pipelines

security and *reliability* by isolating potentially problematic builds.

fault tolerance, the "capability of a product to operate as intended despite the presence of hardware or software faults" *(ISO 2023)*.

interoperability, the "capability of a product to exchange information with other products and mutually use the information that has been exchanged" *(ISO 2023)*.

Satisfies user stories: **US5**, **US8**, **US9**

Figure 4.1: ISO/IEC 25010 is part of the SQuaRE (Software Quality Requirements and Evaluation) series that replaced 9126 *(Wikipedia contributors 2025)*

4.1.4 Non-Functional Requirements

NFR1: RELIABILITY REQUIREMENTS (System)

The Ten build system must function reliably under all specified conditions.

- Must ensure operational stability, handle failures gracefully, maintain data integrity during failures, and provide recovery mechanisms

reliability and *faultlessness*, the "capability of a product to perform specified functions without fault under normal operation" *(ISO 2023)*.

Satisfies user stories: **US1**

NFR2: SECURITY REQUIREMENTS (Privilege Tier)

The system must enforce security boundaries and prevent unauthorised access.

- Must implement privilege separation, validate content integrity cryptographically, isolate builds, and prevent unauthorised store modifications

security and *integrity*

Satisfies user stories: **US2**, **US5**

NFR3: PERFORMANCE REQUIREMENTS (Build)

Efficient performance regarding time and resource usage.

- Must optimize build performance through caching and dependency tracking, use system resources efficiently, scale to large build graphs, and minimize unnecessary rebuilds

performance efficiency and *resource utilisation*

Satisfies user stories: **US4**

NFR4: USABILITY REQUIREMENTS (CLI)

The Ten build system must be learnable and usable by the target audience.

- Must provide clear user interfaces, offer helpful error messages, include comprehensive documentation, and make system state inspectable

learnability, *user assistance*, and *self-descriptiveness*

Satisfies user stories: **US9**

NFR5: COMPATIBILITY REQUIREMENTS (Integration)

Interoperation with existing tools and systems.

- Must co-exist with other package managers, provide integration interfaces, support standard formats, and work across different environments

compatibility and *interoperability*

satisfies user stories: **US2**, **US8**

NFR6: MAINTAINABILITY REQUIREMENTS (Architecture)

The system and the code must be maintainable, modular, and extensible.

- Must structure clear component boundaries, facilitate code reuse, support testing at all levels, enable extension, and provide debugging capabilities

modularity, *reusability*, *analysability*, *testability*, and *modifiability*

Satisfies user stories: **US3**, **US6**, **US8**

5 Design

5.1.0 Reliance on Existing Libraries

Core System Components

- **Core/Standard Libraries**:
  - `base` – Haskell standard library providing fundamental types and functions
  - `text` – Efficient Unicode text handling, essential for package descriptions and output
  - `bytestring` – Fast, compact byte string manipulation used for binary data handling
  - `containers` – Efficient implementations of maps, sets, and sequences for package management
  - `mtl` – Monad transformer library providing the core architectural foundation
  - `exceptions` – Structured exception handling throughout the codebase
- **Filesystem Operations**:
  - `directory` – Directory traversal and manipulation for package store management
  - `filepath` – Path manipulation utilities for store paths
  - `temporary` – Creation and management of temporary files during builds
  - `mmap` – Memory-mapped file I/O for performance optimisation
- **System Integration**:
  - `process` – Process execution capabilities for running build scripts
  - `unix` – Direct access to UNIX system calls for sandbox creation
  - `unix-bytestring` – UNIX-specific bytestring operations
  - `time` – Date and time handling for build logs and metadata
- **Cryptography & Content Addressing**:
  - `cryptonite` – Cryptographic primitives for secure hashing of packages
  - `memory` – Memory manipulation for cryptographic operations
  - `base64-bytestring` – Encoding/decoding for hash representation
- **Database & Storage**:
  - `sqlite-simple` – SQLite interface for package metadata storage
  - `binary` – Binary serialization for efficient data storage

Daemon & Networking

- **Concurrency**:
  - `stm` – Software Transactional Memory for concurrent daemon state management
  - `async` – Asynchronous execution capabilities for parallel builds
- **Network Communication**:
  - `network` – Network communication for client-daemon interactions
- **IPC & Authentication**:
  - `uuid` and `uuid-types` – Generation of unique identifiers for builds and sessions
  - `random` – Random number generation for security tokens

Serialisation & Data Representation

- **Package Description & Configuration**:
  - `aeson` and `aeson-pretty` – JSON parsing and generation for package descriptions
  - `scientific` – Precise representation of numeric values in configurations
- **Data Structures**:
  - `vector` – Efficient array implementation for performance-critical operations
  - `unordered-containers` – Hash-based containers for faster lookups
  - `extra` – Additional utility functions not found in base libraries
- **Type-Level Programming**:
  - `singletons`, `singletons-th`, and `singletons-base` – Advanced type-level programming for enforcing invariants

Potential Future Custom Implementations

Several external libraries could be replaced with custom implementations in future versions:

- **Crypto & Hashing**: While `cryptonite` provides robust implementations, a specialized hashing system tailored specifically to Ten's needs could improve performance.
- **Database Layer**: The `sqlite-simple` dependency introduces an external dependency that could be replaced with a custom storage solution optimised for Ten's access patterns.
- **JSON Parsing**: The `aeson` library is comprehensive but may include more functionality than needed. A lightweight parser focused only on Ten's package description format could reduce overhead.
- **Sandbox Implementation**: Currently using Unix primitives through the `unix` library, but a more refined sandbox implementation could provide better isolation and security.
- **Build Scheduler**: The current implementation uses `async` for parallelism, but a custom scheduler could better optimise resource usage during complex builds.

Testing Infrastructure

The test suite leverages standard Haskell testing libraries:

- `HUnit` – Unit testing framework for testing individual components
- `hspec` – Behaviour Driven Development-style testing for higher-level system behavior
- `QuickCheck` – Property-based testing for verifying invariants across arbitrary inputs

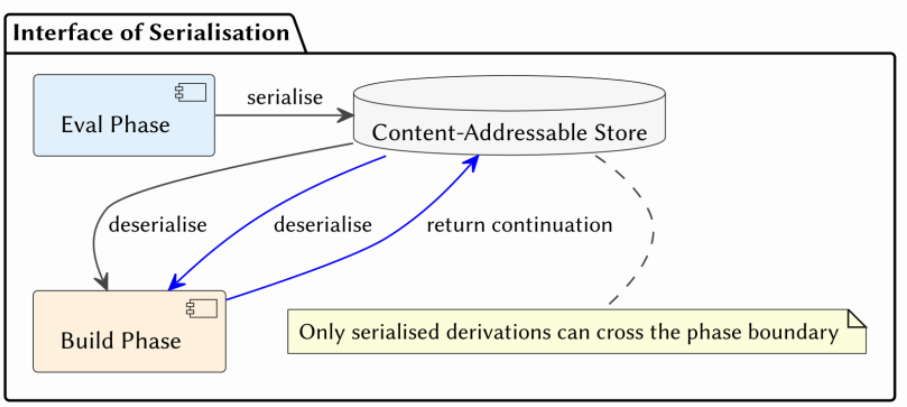
Ten balances the use of existing, well-tested libraries for core functionality with the option to replace them with specialised implementations as the project matures beyond the MVP stage. The current dependency selection focuses on reliability and development speed while ensuring the system meets its requirement of functional coherence on Linux.

5.2.0 Modular Architecture

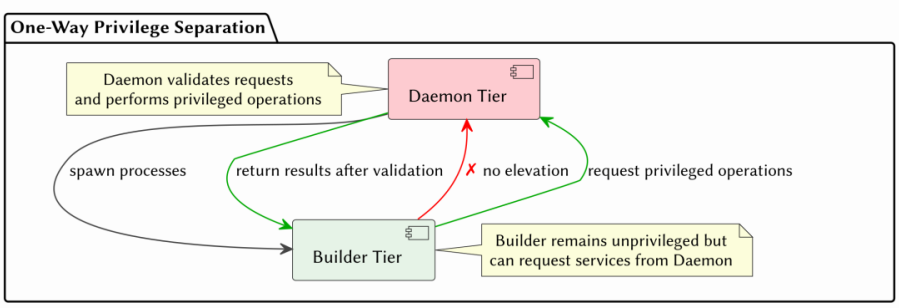
```
Core System (Foundation)

src/Ten/Core.hs - Fundamental type system, monad transformations, and privilege contexts
src/Ten/Store.hs - Store path and content operations
src/Ten/Hash.hs - Content addressing and hash functions

Protocol Layer (Communication Boundary)

src/Ten/Daemon/Protocol.hs - Protocol definitions, message types, and serialisation
src/Ten/Daemon/Auth.hs - Authentication and authorisation between contexts

Database Layer (Privileged Operations)

src/Ten/DB/Core.hs - Database access primitives and connection handling
src/Ten/DB/Schema.hs - Schema definitions with privilege considerations
src/Ten/DB/Derivations.hs - Derivation meta-data storage with context awareness
src/Ten/DB/References.hs - Reference meta-data tracking with privilege checks

Build System (Split Implementation)

src/Ten/Derivation.hs - Derivation definition and operations
src/Ten/Graph.hs - Dependency graph handling
src/Ten/Sandbox.hs - Isolated build environment with privilege separation
src/Ten/Build.hs - Core build logic with context awareness
src/Ten/GC.hs - Garbage collection with proper privilege handling

Daemon Implementation (Privileged Services)

src/Ten/Daemon/State.hs - Daemon state management
src/Ten/Daemon/Server.hs - Server implementation handling builder requests
src/Ten/Daemon/Client.hs - Client implementation for unprivileged operations

TO DO: Integration and Entry Points <NOT IMPLEMENTED>

-- src/Ten.hs - Main library module
-- src/Ten/CLI.hs - Command-line interface
-- app/Main.hs - Client executable entry point
-- app/TenDaemon.hs - Daemon executable entry point
```

5.3.0 Ten's Inviolable Boundaries: Phases and Tiers

- **Orthogonal Type-Level Labelling, Explicit Context Tracking**: Phase and Privilege dimensions are separate type parameters, creating a two-dimensional context matrix. Every operation is aware of both its phase and privilege context
- **Static Type-Level Distinctions**: Use types to prevent invalid operations (build-time operations during evaluation or privileged operations in unprivileged contexts)

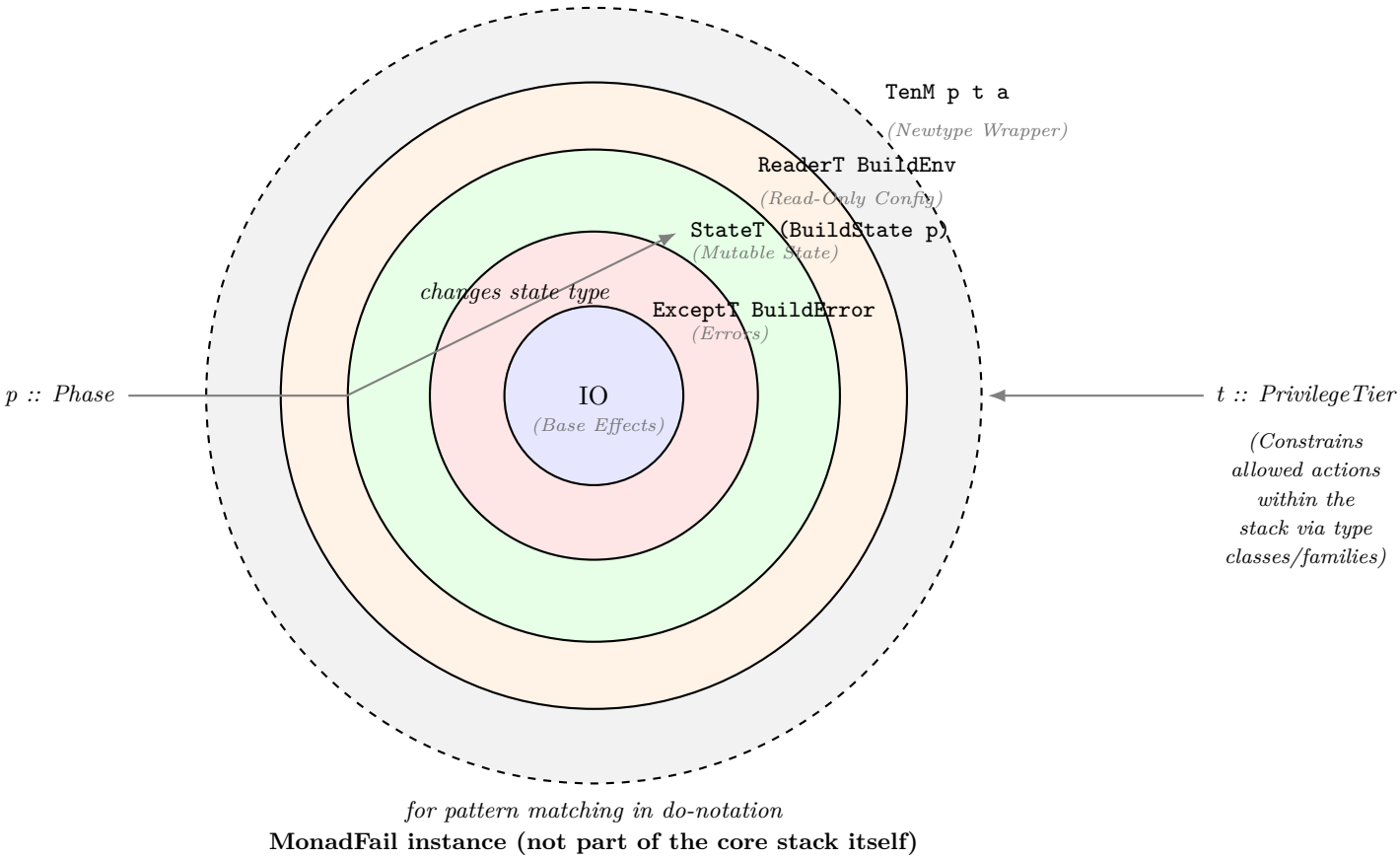


- **Deterministic Evaluation**: Eval phase guarantees consistent derivation graphs regardless of environment
- **Derivation as Interface**: Derivations form the boundary between evaluation and build phases the "Interface of Serialisation".
- **Acyclic Dependency Verification**: Validate graph acyclicity during evaluation
- **Build Strategy Inference**: Automatically determine optimal build strategy (applicative vs monadic)
- **Return-Continuation Pattern**: Support multi-stage builds through chained derivations
- **Pure Evaluation Environment**: Evaluation context contains no system state, only pure dependencies



- **Protocol-Based Privilege Boundary**: Communication between privileged and unprivileged code goes through an authenticated protocol
- **One-Way Privilege Separation**: Operations requiring privilege are requested from the Daemon. There is no requirement or implementation for transition or elevation of privilege.
- **Store-as-a-Service**: Content-addressable store operations are services provided by daemon to builders
- **Complete Sandbox Isolation**: Builders run in isolated environments with controlled access to resources

5.4.0 The Monad Transformer Stack



- Parameterised Monad (TenM): Indexed by Phase (Eval, Build) and PrivilegeTier (Daemon, Builder) to enforce constraints at the type level.
- Singletons (SPhase, SPrivilegeTier): To bridge the gap between type-level constraints and runtime execution, ensuring type safety.
- Type Families (CanAccessStore, etc.) and Constraints (RequiresDaemon, etc.): To formally define and check capabilities based on the type parameters.
- Transformer Stack: ReaderT, StateT, ExceptT, IO to manage environment, state, errors, and side effects cleanly.
- Rich Data Types: Representing the domain concepts like Derivation, StorePath, BuildResult, BuildEnv, BuildState, protocol messages, etc.
- Privilege-Specific Implementations: Functions like storeDerivationDaemon vs. storeDerivationBuilder showcase how actions differ based on the PrivilegeTier

5.5.0 API Concept

Ten features a singular, unified, functional, cross-paradigm syntax.

```
-- Derivations are serialised for storage in ten/store/
ten-instantiate myDerivation.ten

-- Serialised build recipe in store can be executed.
ten-realise \
  /ten/store/1a2b3c4d5e6f7g8h9i0j1k2l3m4n5o6p-myDerivation.drv

-- Alternatively, both actions can be undertaken simultaneously:
ten-build myDerivation.ten
```

Applicative (Parallel) Builds

```
-- Ten automatically detects these are independent and builds them in
    parallel, despite the do-notation.

buildParallel = do
  lib <- derivation { name = "lib"; ... }
  docs <- derivation { name = "docs"; ... }
  tests <- derivation { name = "tests"; ... }

  return { library = lib; documentation = docs; tests = tests }
```

Monadic (Sequential) Builds

```
-- Ten infers from structure the dependency chain and ensures sequential
    building.

buildSequential = do
  lib <- derivation { name = "library"; ... }
  app <- derivation {
    name = "application";
    builder = "${gcc}/bin/gcc";
    args = ["-L${lib}/lib", "-o", "$out/bin/app", "main.c"];
  }

  return app
```

Return-Continuation (Recursive) Builds

```
bootstrapCompiler = do
  result <- derivation {
    name = "compiler-bootstrap";
    builder = "${gcc}/bin/gcc";
    args = ["-o", "$out/bin/compile", "compiler.c"];
  }

  return result
```

6 Results

6.1.0 Requirements Traceability Matrix

ID	Requirement	Component	Status
FR1: Pure Functional Core Requirements			
FR1.1	Referential Transparency	Core	Satisfied
FR1.2	Immutable Store	StorePath	Satisfied
FR1.3	Declarative Specifications	DSL	Unsatisfied
FR2: DSL Interface for Build Specification			
FR2.1	Ten Expression Language	Parser	Inconclusive
FR2.2	Functional Path References	StorePath	Satisfied
FR3: Holistic Executive Cohesion of Modular Architecture			
FR3.1	Pure Functional Builds	Build	Satisfied
FR3.2	Content-Addressed Storage	Hash	Satisfied
FR3.3	Immutable Store	StorePath	Satisfied
FR3.4	Garbage Collection	GC	Inconclusive
FR3.5	Layered Architecture	Monad Stack	Satisfied
FR3.6	Daemon-Based Operation	Privilege Tier	Inconclusive
FR4: Derivation Management			
FR4.1	Derivation Representation	Derivation	Satisfied
FR4.2	Dependency Resolution	Graph	Inconclusive
FR4.3	Build Strategies	Derivation	Satisfied
FR5: Store Management			
FR5.1	Store Path Validation	StorePath	Satisfied
FR5.2	Hash Verification	Hash	Satisfied
FR5.3	Content Deduplication	Store	Inconclusive
FR6: Error Handling & System Integration			
FR6.1	Sandbox Isolation	Sandbox	Satisfied
FR6.2	Error Propagation	Build	Satisfied
FR6.3	System Integration	CLI	Unsatisfied
Non-Functional Requirements			
NFR1	Reliability Requirements	System	Unsatisfied
NFR2	Security Requirements	Privilege Tier	Inconclusive
NFR3	Performance Requirements	Build	Inconclusive
NFR4	Usability Requirements	CLI	Unsatisfied
NFR5	Compatibility Requirements	Integration	Unsatisfied
NFR6	Maintainability Requirements	Architecture	Satisfied

Analysis

The Requirements Traceability Matrix demonstrates that the Ten project has successfully implemented its core architectural foundation while several planned components remain either unimplemented or have inconclusive testing results.

Satisfied Requirements (11/22):

- **Core Storage Functionality:** All key storage components (StorePath, Hash, Immutable Store) have been successfully implemented and verified through tests. This forms the foundation of the content-addressable store which is central to the Ten architecture.
- **Build System Core:** The fundamental build mechanisms (Pure Functional Builds, Derivation Representation, Build Strategies) are operational and pass their associated tests. These components enable deterministic build outputs.
- **Architecture Patterns:** The Layered Architecture with the Monad Transformer Stack is properly implemented, providing type-level separation of concerns and the foundation for privilege restriction.
- **Error Management:** Error propagation and sandbox isolation functions are working correctly, ensuring build failures are properly captured and communicated.

Unsatisfied Requirements (6/22):

- **User Interface Layer:** The CLI and system integration components have not been implemented, meaning the system cannot yet be used through a command-line interface.
- **DSL Implementation:** The Ten Expression Language and declarative specifications are not yet implemented, limiting the ability to write package definitions in a user-friendly way.
- **System-Level Requirements:** Overall reliability, usability, and compatibility requirements remain unsatisfied due to their dependence on the unimplemented components.

Inconclusive Requirements (7/22):

- **Privilege Separation:** While the type-level privilege restriction mechanism exists in the architecture, tests for components that rely on it (such as Graph operations) are failing with "Privilege tier mismatch" errors. This indicates the mechanism is detecting operations performed in inappropriate contexts, but it's unclear whether this is due to test design issues or implementation problems.
- **Advanced Store Features:** Garbage collection and content de-duplication have implementations but lack conclusive tests to verify their correctness.
- **Graph Operations:** Dependency resolution tests are failing, though the core implementation exists. This is a critical component for resolving build dependencies.
- **Performance and Security:** These non-functional requirements have partial implementations but insufficient test coverage to determine their status conclusively.

Notably, the successful compilation of the entire codebase is itself a significant test that validates the type safety mechanisms are working properly. The Privilege Tiers and Phases have been integrated across the architecture, as evidenced in modules like Ten.Daemon.Server, Ten.Sandbox, and Ten.GC (among others). The type system is correctly enforcing proper privilege boundaries through the TenM monad transformer stack, demonstrating that key architectural requirements are being met even though some test cases are failing.

The current implementation validates the content-addressable store, derivation handling, and build verification components working in practice. The privilege separation mechanism is showing signs

of functioning at the type level by preventing inappropriate operations, even though this is causing test failures. Build success indicates that Phases and Tiers are implemented adequately and fulfilling their role in enabling system-wide type-harmony for the rest of the implementation.

Ultimately, the project fails to deliver on its promises but falls within a hairs-breadth of its key goals. It may come to light in the coming months just how close it came before the deadline but, alas, there really is no more time before submission. Still, the concepts and dynamics that have been verified, and those left as yet unproven in the codebase can stand on their own two feet. The key steps from here will be universal test suite integration, bolstering the existing implementation and highlighting the requirements to take this project from here to finality for its first release.

6.2.0 Ten MVP - Monad Architecture Validation

In the ten-mvp repo (https://github.com/enzojoly/ten-mvp) we emulate the dynamic multi-layered context of the monad transformer architecture and apply generative testing for property-based verification and proof of all MonadT functionality working as expected.

```
Build profile: -w ghc-9.8.2 -O1
In order, the following will be built (use -v for more details):
 - ten-mvp-0.1.0.0 (test:ten-mvp-test) (first run)
Preprocessing test suite 'ten-mvp-test' for ten-mvp-0.1.0.0...
Building test suite 'ten-mvp-test' for ten-mvp-0.1.0.0...
Running 1 test suites...
Test suite ten-mvp-test: RUNNING...

TenM ('Eval 'Daemon)
  Functor Laws
    Identity (pure): fmap id . pure === pure [✓]
      +++ OK, passed 100 tests.
    Identity (error): fmap id . throwError === throwError [✓]
      +++ OK, passed 100 tests.
    Composition (pure): fmap (f . g) . pure === (fmap f . fmap g) . pure [✓]
      +++ OK, passed 100 tests.
    Composition (error): fmap (f . g) . throwError === (fmap f . fmap g) . throwError [✓]
      +++ OK, passed 100 tests.
  Applicative Laws
    Identity: pure id <*> v === v (testing with pure) [✓]
      +++ OK, passed 100 tests.
    Identity: pure id <*> v === v (testing with error) [✓]
      +++ OK, passed 100 tests.
    Homomorphism: pure f <*> pure x === pure (f x) [✓]
      +++ OK, passed 100 tests.
    Interchange: u <*> pure y === pure ($ y) <*> u (testing with u = pure f) [✓]
      +++ OK, passed 100 tests.
    Interchange: u <*> pure y === pure ($ y) <*> u (testing with u = error) [✓]
      +++ OK, passed 100 tests.
    Composition (pure): pure (.) <*> pure f <*> pure g <*> pure x === pure f <*> (pure g <*> pure x) [✓]
      +++ OK, passed 100 tests.
    Composition (error left): pure (.) <*> error <*> v <*> w === error <*> (v <*> w) [✓]
      +++ OK, passed 100 tests.
    Composition (error middle): pure (.) <*> u <*> error <*> w === u <*> (error <*> w) [✓]
      +++ OK, passed 100 tests.
    Composition (error right): pure (.) <*> u <*> v <*> error === u <*> (v <*> error) [✓]
      +++ OK, passed 100 tests.
    fmap f x === pure f <*> x (pure) [✓]
      +++ OK, passed 100 tests.
    fmap f x === pure f <*> x (error) [✓]
      +++ OK, passed 100 tests.
  Monad Laws
    Left Identity (k returns pure): pure a >>= (pure . k') === pure (k' a) [✓]
      +++ OK, passed 100 tests.
    Left Identity (k throws error): pure a >>= (throwError . k_err) === throwError (k_err a) [✓]
      +++ OK, passed 100 tests.
    Right Identity: m >>= pure === m (testing with pure) [✓]
      +++ OK, passed 100 tests.
    Right Identity: m >>= pure === m (testing with error) [✓]
      +++ OK, passed 100 tests.
    Associativity (pure >>= (pure . f') >>= (pure . g')) [✓]
      +++ OK, passed 100 tests.
    Associativity (error >>= (pure . f') >>= (pure . g')) [✓]
      +++ OK, passed 100 tests.
    Associativity (pure >>= (throwError . f_err) >>= (pure . g')) [✓]
      +++ OK, passed 100 tests.
    Associativity (pure >>= (pure . f') >>= (throwError . g_err)) [✓]
      +++ OK, passed 100 tests.

Finished in 0.0290 seconds
23 examples, 0 failures
Test suite ten-mvp-test: PASS
Test suite logged to:
/home/enzo/_ten/ten-mvp/dist-newstyle/build/x86_64-linux/ghc-9.8.2/ten-mvp-0.1.0.0/t/ten-mvp-test/test/ten-mvp-0.1.0.0-ten-mvp-test.log
1 of 1 test suites (1 of 1 test cases) passed.
```

6.3.0 HUnit, HSpec, QuickCheck - Test Results Summary

We use QuickCheck for generative testing of system invariants, and HUnit for specified unit testing for specific functionality. The test output shows mixed results:

Passing Tests:

- All StorePath unit tests and properties (7 tests)
- All Hash tests (2 tests)
- All Derivation tests (2 tests)
- All Sandbox tests (1 test)
- All Build tests (1 test)

Failing Tests:

- All Graph unit tests (2 tests)
- All Graph property tests (2 tests)

The failing tests all show a similar error pattern: PrivilegeError "Privilege tier mismatch".

This suggests either that the tests are failing wrongly, or that the tests require a rewrite enabling some workaround ad hoc privilege in order to test functionality properly.

Test Categories

The tests cover several critical aspects of the Ten build system:

- StorePath Tests: Validating path format, hash validation, and path conversion
- Hash Tests: Ensuring hashing functions produce consistent and unique outputs
- Graph Tests: Testing dependency graph operations (cycle detection and topological sorting)
- Derivation Tests: Verifying serialisation/deserialisation and equality comparison
- Sandbox Tests: Testing configuration for isolated build environments
- Build Tests: Verifying build result validation

7 Reflection

7.1.0 Mezirow's Transformative Learning Theory

As I reflect on this project, I'm reminded that software development is a human endeavour, and I'd first like to express my gratitude to the people who've made it possible. My supervisor Ryan for the constant support and encouragement, and my tutor Steve for the same energy and support, family, friends, Sphere, and Diego - all have contributed to this project's success. It's a miracle to have finished it within the deadline, without a budget, and at times what felt like without a prayer, to be perfectly candid!

'Success' in this case relies entirely on the prospective viewpoint of the beholder. Success is subjective, and from my vantage point, it's a complex landscape. I'm still new to development, and this project was an ambitious undertaking. I'm thankful I didn't opt for the RISC-V processor project, though maybe I would have found a way to succeed regardless. I've worked tirelessly for seven, eight, or even nine months, pulling all-nighters and sacrificing time with loved ones. The results are good, but not complete a theme that may continue in the future. Does software ever truly get finished?

Even if I've fulfilled 90% of this project, the first 90% is only the first 50% in reality or so they say; such a token feels pertinent at this time. I feel a sense of guilt for running out of time, but I can also admit that I have grown from this experience. These Mezirow cycles are fractal, occurring daily, and compounding, spreading over the course of years. I've come to see that every day is a challenge and a new opportunity to set sail on the open ocean. My attitudes to development have changed, as they tend to do from time to time. Of all things, I've certainly learned to appreciate the journey.

We can also consider that we define and design our daily trials as humans. I have known that I am responsible for taking my own life by the reigns for some years now, and some of us make our own problems. I am a keen believer in the growth that comes following subjecting oneself to hardship. Not cruelty or needless suffering but orchestrated difficulty of a load that is just beyond the limits we can bear as professionals, as technicians and ultimately as human beings. Impostor Syndrome runs deep. Our own reality can become little more than a figment of self-delusion, sometimes. It is often self-constructed. The truth to Mezirow's Transformative Learning Theory is that the real solution to the petulant torrents of life seems to be a deep inner meditation and calm, without delusion, facing that which seems daunting at first. This allows us to reflect, find new perspective, and quite literally think our way critically out of our own problems (which is not always possible to do). I am by no means unshakeable. But every day I try to decompress and reflect on that which shook me from my resolve. And I may hope for the days which nothing shakes me, and yet the days we grow the most as people are arguably those days of stress and torment. In terms of the project, I of course had to strike a precise balance of what was too little or too much work to undertake, and that was perhaps the most difficult element to this module. My ambition mixed with my drive

to do just a little bit more. Slow tides that calm raging rapids or some such metaphor to that effect.

Truly I am disappointed in the finished product of my work, or perhaps that I did not get to finish it. But I am fine with that. Sometimes you must draw a line under something. If we do not compose - finish - our compositions, there will be no pieces of music, only a flurry of fatigued perfectionists rattling instruments. And at least it was not a failure. I will work to improve on it in due time. The piece of work I am least proud of is this report. I had such a vision for what it would be. And the thing about delusions is that they do not fade softly, they are shattered violently when reality hits.

And so, despite the fact I am not really proud of the finished state of my project, nor do I feel it would be right to be "proud" per se, I am proud that it exists, and that it is *my* project and I know it has the potential to be something I am most proud of altogether, not just for what it is, but for what it symbolises as well.

7.2.0 Conclusion

The project successfully achieves its core objectives:

- Implementing a content-addressable store with integrity verification
- Conceptualising a pure functional DSL for package definitions
- Providing atomic, transactional builds with isolation
- Supporting dynamic dependencies through the return-continuation pattern
- Enforcing security boundaries through type-level privilege separation

Future work:

- Codified build recipe DSL implementation (from concept)
- Exhaustive Test Suite integration
- Command-line interface
- Optimisation, custom implementations for imported libraries
- Visualisation mode for operational demos?
- QOL features emulating Nix?

While Ten is primarily a research prototype rather than a production-ready system, it demonstrates the potential of applying advanced type theory to systems programming challenges. The lessons learned from this project can inform the design of future build systems and package managers, particularly in contexts where correctness guarantees are paramount.

To some degree, the success of Ten shows that Haskell's type system and pure functional approach are powerful tools for building reliable system software. By encoding invariants in the type system, Ten demonstrates how software can be designed to be correct by construction, reducing the burden of testing and manual verification.

Bibliography

Akerholt, G., K. Hammond, and S. Peyton Jones (1991). "Title of chapter". In: *Functional Programming, Glasgow 1991*. Ed. by R. Heldal, C.K. Holst, and P. Wadler. Workshops in Computing. Springer.

Bird, R. and P. Wadler (1988). *Introduction to Functional Programming*. Prentice Hall International Series in Computing Science. Prentice Hall.

Dahl, V. and P. Wadler, eds. (2003). *Practical Aspects of Declarative Languages: 5th International Symposium, PADL 2003*. Vol. 2562. Lecture Notes in Computer Science. New Orleans, LA, USA: Springer.

Dolstra, E. (2006). *The Purely Functional Software Deployment Model*. Utrecht University.

Garrity, T.A. (2021). *All the Maths You Missed*. 2nd. Cambridge University Press.

Gay, W. (2000). *Linux Socket Programming by Example*. Que.

Hagiya, M. and P. Wadler, eds. (2006). *Functional and Logic Programming: 8th International Symposium, FLOPS 2006*. Vol. 3945. Lecture Notes in Computer Science. Springer.

Hasegawa, M., ed. (2013). *Typed Lambda Calculi and Applications: 11th International Conference, TLCA 2013*. Vol. 7941. Lecture Notes in Computer Science. Eindhoven, The Netherlands: Springer.

Heldal, R., C.K. Holst, and P. Wadler, eds. (1991). *Functional Programming, Glasgow 1991*. Workshops in Computing. Springer.

ISO (2023). *ISO/IEC 25010:2023 Systems and software engineering Systems and software Quality Requirements and Evaluation (SQuaRE) Product quality model*. International Standard ISO/IEC 25010:2023. International Organization for Standardization.

Milewski, B. (2025). *The Dao of Functional Programming*. Accessed: 2 April 2025. URL: https://github.com/BartoszMilewski/Publications/blob/master/TheDaoOfFP/DaoFP.pdf.

Peyton Jones, S.L. and D.R. Lester (2000). *Implementing functional languages: a tutorial*. Web draft.

Peyton-Jones, S. (2013). "Title of chapter". In: *Typed Lambda Calculi and Applications: 11th International Conference, TLCA 2013*. Ed. by M. Hasegawa. Vol. 7941. Lecture Notes in Computer Science. Springer.

Sannella, D. et al. (2020). *Introduction to Computation: Haskell, Logic and Automata*. Undergraduate Topics in Computer Science. Springer International Publishing.

Skinner, R. (Aug. 2023). *Effective Haskell: Solving Real-World Problems with Strongly Typed Functional Programming*. 1st ed. Pragmatic Bookshelf. ISBN: 9781680509342.

Wadler, P. (1992). *The essence of functional programming*. Technical Report. University of Glasgow.

— (2001). *Monads for functional programming*. Technical Report. University of Glasgow.

— (2004). *Comprehending Monads*. Technical Report. University of Glasgow.

Wadler, P. and R.B. Findler (2009). "Title of chapter". In: *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009*. Ed. by G. Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer.

Wikipedia contributors (2025). *ISO/IEC 9126*. Accessed: 9 May 2024. URL: https://en.wikipedia.org/wiki/ISO/IEC_9126.

A

Glossary

ApplicativeStrategy A build strategy for derivations with statically known dependencies, enabling concurrent building and optimisation. Unlike MonadicStrategy, ApplicativeStrategy allows more efficient parallelisation since all dependencies are known before execution begins.

AST (Abstract Syntax Tree) A tree representation of the abstract syntactic structure of the Ten expression language code. Generated by the Parser component from .ten files, with nodes representing expressions, literals, and control structures in a hierarchical form.

Atomicity A property ensuring that build operations are treated as a single, indivisible unit that either completes entirely or not at all. Ten implements atomicity through file system operations and database transactions that maintain system consistency even in the event of crashes.

Build Environment The isolated context in which a build process executes, containing all necessary dependencies and configurations. In Ten, each build has its own environment with controlled access to inputs, environment variables, and system resources.

Build Monad (TenM) The monad transformer stack providing the core computational context for the build system, combining pure evaluation with controlled effects. Parameterised by both Phase and PrivilegeTier type variables to enforce separation at the type level.

Build Function (BuildFn) A pure function that executes derivations to produce outputs in a deterministic manner. In Ten, BuildFn is implemented with proper privilege separation between the daemon and builder contexts.

Build Recipe A declarative specification that defines how to build a package, including its dependencies, build steps, and configurations. Written in Ten Expression Language and evaluated to produce concrete derivations.

Build Sandbox An isolated environment that restricts network and file system access to ensure builds are deterministic and reproducible. Implemented using Linux namespaces, bind mounts, and capability restrictions to create hermetic environments.

BuildStatus An enumeration tracking the state of a build: Pending, Running (with progress percentage), Recursing (to another derivation), Completed, or Failed. Maintained by the daemon and queryable by clients.

Cache Hit When a requested build output is found in the Store, allowing reuse without rebuilding. Ten identifies cache hits by comparing cryptographic hashes of derivation inputs against stored outputs.

Cache Logic Component that determines whether a derivation's output exists in the Store or needs to be built. The daemon implements this logic by checking for store paths that match the expected hash of outputs.

Cache Miss When a requested build output is not found in the Store, requiring a new build. Ten handles cache misses by spawning builder processes in isolated sandboxes to create the missing outputs.

Content-Addressable Storage A method of storing information that can be retrieved based on its content rather than its location. Ten implements this with store paths derived from SHA256 hashes, enabling integrity verification and de-duplication.

DaemonConnection A handle to communicate with the Ten daemon process, including authenticated channels for requesting builds and store operations. Manages message framing, serialisation, and response handling.

DaemonRequest/DaemonResponse The protocol messages exchanged between clients and the daemon, encompassing operations like building, store access, and garbage collection. Structured with proper type safety and serialisation.

Derivation A pure functional description of a build action, including its inputs, build steps, and expected outputs. The fundamental unit of build specification in Ten, represented as immutable, serialisable records with cryptographic hashing.

DerivationInput A reference to a store path that serves as an input to a derivation, including both the path and a name used to access it during the build. Inputs may come from previous builds or external sources.

DerivationOutput An expected output from a derivation, including its name and the store path where it will be created. Ten supports multiple named outputs from a single derivation.

Dependency Graph (DGraph) A directed acyclic graph (DAG) representing relationships between different derivations and their dependencies. Ten uses this graph for topological sorting, cycle detection, and parallel build scheduling.

Deterministic Build A build that produces identical outputs given the same inputs, regardless of when or where it runs. Ten enforces determinism through sandboxing, explicit dependencies, and controlled execution environments.

DSL (Domain-Specific Language) The specialised language (Ten Expression Language) used to write build specifications and configurations. A pure functional language with features for dependency management, build description, and system configuration.

Effect Layer The system layer that handles interactions with the outside world, including file system operations and build execution. Ten strictly separates effects from pure evaluation to maintain determinism.

Evaluator Component that processes the AST to produce derivations, creating temporary GC roots for build outputs. The evaluator operates in the pure Eval phase, before any actual building occurs.

Free Monad A way to represent program structure separately from its interpretation, used in the DSL implementation. Ten uses Free Monads to define build operations independently from their execution strategy.

Functional Purity The property that a function's output depends only on its inputs, without side effects or hidden dependencies. Ten enforces purity in its evaluation phase to enable reliable caching and parallelisation.

Garbage Collector (GC) Component that removes unreferenced paths from the Store to free up space. Ten's GC traces references from roots, computes reachability, and safely removes unreachable artifacts.

GC Roots References that prevent the Garbage Collector from deleting certain paths in the Store. Ten maintains different types of roots including runtime dependencies, build outputs, and explicitly registered entries.

Hash Function (HashFn) Function that computes unique identifiers for derivations based on their content and dependencies. Ten uses SHA256 for strong cryptographic hashing throughout the system.

Immutability The property that once created, build outputs cannot be modified, only replaced with new versions. Ten enforces this through read-only permissions and content-addressed storage paths.

IO Effects Side effects involving input/output operations that must be carefully controlled and isolated in the build system. Ten handles these through privilege separation and the TenM monad.

LLM Large Language Model, a type of neural network architecture based primarily on transformer mechanisms (typically decoder-based), trained on vast text corpora with billions of parameters. These models can generate human-like text, respond to queries, and perform various language tasks. Examples include GPT, Claude, and LLaMA.

Monad Transformer A type constructor that adds capabilities to an existing monad, used to compose different effects in the build system. Ten uses ReaderT, StateT, and ExceptT to construct its core TenM monad.

MonadicStrategy A build strategy for derivations with dynamic dependencies or those using the Return-Continuation pattern. Unlike ApplicativeStrategy, builds must be executed sequentially as dependencies might only be known (or knowable) (or knowable) during execution.

NLP Natural Language Processing, a field of computer science, artificial intelligence, and linguistics focused on enabling computers to understand, interpret, and generate human language. It encompasses various techniques including statistical methods, machine learning, and rule-based approaches.

Parser Component that reads .ten files and converts them into an Abstract Syntax Tree (AST). Implements the grammar of the Ten Expression Language, handling syntax checking and error reporting.

Phase A type-level tag indicating whether operations are in the Eval (analysis) or Build (execution) phase. Ten uses phantom types to enforce phase separation, preventing build-time operations during evaluation.

Phase Separation The strict separation between evaluation time (when build plans are created) and build time (when they are executed). Ten's functional adherence to the notion of Disallow Import From Derivation.

PLT Programming Language Theory.

PrivilegeTier A type-level tag indicating whether operations run at the Daemon (privileged) or Builder (restricted) level. Ten uses this to enforce security boundaries at compile-time.

Protocol The message-based communication system between the daemon and clients. Includes serialisation, authentication, and framing of requests and responses with proper error handling.

Pure Core The functional core of the system where all operations are pure functions without side effects. Ten's Eval phase operates in this pure context, enabling reasoning and optimisation.

ReaderT A monad transformer providing read-only access to shared environment configuration throughout the build process.

Referential Transparency The property that an expression can be replaced with its value without changing programme behaviour. Ten should enforce this in its expression language to enable caching and optimisation.

Return-Continuation A pattern where a build produces another derivation to be built rather than final outputs directly. Ten supports this pattern for multi-stage builds like bootstrapping compilers.

SandboxCreator The component responsible for creating isolated build environments with proper restrictions. Ten implements different versions for privileged daemon operations versus restricted builder contexts.

StateT A monad transformer for maintaining mutable state in a pure way during builds. Ten uses this to track build progress, collect outputs, and manage resource allocation.

Store The content-addressable storage system that holds all build outputs and dependencies. Implemented as a special directory with paths derived from content hashes, ensuring integrity and enabling deduplication.

Store Path A unique path in the Store that is derived from the hash of its contents and dependencies. Ten paths follow the pattern '/store/hash-name' where the hash guarantees uniqueness and integrity.

Ten Expression Language The pure functional DSL used to write build specifications in .ten files. Features include lazy evaluation, higher-order functions, and declarative dependency management. See above: DSL (Domain-Specific Language).

Thunk A suspended computation that is evaluated only when its result is needed, used for lazy evaluation in the build system.

Transactional Property ensuring that a series of operations either completes entirely or has no effect, maintaining system consistency. Ten implements this for store operations and database interactions.

Type-Level Programming Using Haskell's type system to enforce constraints and guarantee properties at compile time. Ten uses advanced type features to prevent phase confusion and privilege escalation statically.

Validation The process of checking build inputs, outputs, and intermediate states for correctness and consistency. Ten performs validation at multiple levels, from hash verification to privilege checks.