a permanent, immutable artifact storage system that guarantees hermetic, reproducible builds for decades into
less cross-team collaboration through shared artifacts and automated consumer validation for safe dependency
ust be explicitly version-controlled, allowing developers to git clone a repository, fetch its toolchain and other
nd build successfully without any internet connectivity.

hosting built artifacts, which initially appeared to offer several advantages:

access control mechanisms
their own artifact destiny
CD pipelines
e

acts in GitLab, any project restructuring becomes extremely problematic:

all downstream dependency links
GitLab hierarchy invalidates artifact URLs
re essentially impossible without breaking all consumers
ces create tight coupling to repository structure

ion, violating the fundamental requirement that builds must be reproducible
ete artifacts, all downstream builds that depend on those artifacts become irreparably broken
rmanence undermines the ability to rebuild historical versions
an silently delete critical dependencies

acts multiple times across different projects
ubuntu:20.04 container to their registries, it's stored 20 times in S3
orage means identical files consume storage proportional to the number of copies
y with team adoption rather than with unique content
ompilers, base images, libraries) are duplicated hundreds of times

d support for package formats
packages, forcing use of generic package registries
hat-specific features and metadata
quire different registry configurations

sk identified for dependency on COTS tools like GitLab
would require extensive artifact migration

- No clear path to preserve artifact URLs and references during migration
- Vendor-specific features create additional lock-in

- No visibility into who consumes published artifacts
- Teams cannot assess the impact of updates on downstream consumers
- No automated mechanism to validate that updates won't break dependent projects
- Difficult to discover and reuse artifacts from other teams
- No way to notify consumers when new versions are available

Orchard addresses these fundamental problems through:

1. **Content-Addressable Storage:** Artifacts are referenced by their content hash, not URLs, making project reorganization painless
2. **Automatic Deduplication:** Identical content is stored only once regardless of how many teams use it, dramatically reducing storage costs
3. **Immutable Storage**: Artifacts can never be deleted, only access-controlled, guaranteeing reproducibility
4. **Format Agnostic:** Supports all package formats uniformly while maintaining format-aware capabilities
5. **Vendor Independent**: Uses S3-compatible storage as a backend, avoiding proprietary lock-in
6. **Air-Gap Ready:** Designed for environments without internet access, supporting export/import for disconnected operations
7. **Cross-Team Collaboration**: Enables teams to discover and utilize builds from other teams, fostering organizational reuse
8. **Consumer Tracking**: Automatically identifies all consumers of published artifacts, providing visibility into dependency chains
9. **Automated Update Validation**: Tests proposed updates against all known consumers, ensuring changes can be safely applied before merge
10. **20-Year Reproducibility:** Every design decision prioritizes the ability to reproduce builds decades into the future

Orchard transforms how teams share and consume artifacts:

- **Discovery**: Teams can easily find and reuse artifacts published by other teams
- **Impact Analysis:** Publishers can see exactly who depends on their artifacts before making changes
- **Automated Safety:** When new versions are published, Orchard automatically creates merge requests in consumer projects and validates compatibility
- **Dependency Transparency:** Complete visibility into the entire organizational dependency graph
- **Proactive Updates:** Consumers are automatically notified of updates and security patches in their dependencies

Orchard is a centralized binary artifact storage system that provides content-addressable storage with automatic deduplication, flexible access control, multi-format package support, and dependency tracking capabilities for automated consumer updates. Like an orchard that cultivates and distributes fruit, Orchard nurtures and distributes the products of software builds. The system is designed to support air-gapped environments and ensure fully reproducible builds without requiring internet access.

- The system SHALL store all artifacts using their SHA256 hash as the primary identifier
- The system SHALL automatically deduplicate identical content regardless of filename or upload source
- The system SHALL maintain reference counting for all stored content
- The system SHALL accept any file type and size without restriction
- **The system SHALL permanently retain all uploaded content - deletions are never permitted**
- The system SHALL support "soft deletion" that removes user access while preserving the underlying data
- The system SHALL maintain an audit trail of all content access and visibility changes

- The system SHALL use S3-compatible object storage exclusively as the backend
- The system SHALL support any S3-compatible API (AWS S3, MinIO, Ceph, etc.)
- The system SHALL support custom S3 endpoints for private/air-gapped deployments
- The system SHALL implement S3 multipart upload for large files
- The system SHALL use S3 server-side encryption when available

- The system SHALL organize content into a hierarchy: **Grove** (Project) → **Tree** (Package) → **Fruit** (Instance)
- The system SHALL support multiple tagged references pointing to the same fruit (instance)
- The system SHALL use the URL pattern `/grove/{project}/{tree}/+/{ref}` for artifact access
- The system SHALL support reference types including tags, versions, and direct fruit IDs
- The system SHALL maintain metadata for each harvest (upload) including timestamp, harvester, and original filename

```
/grove/blinx-core/kernel/+/latest
/grove/blinx-core/rootfs/+/version:2024.1
/grove/components/rtc-driver/+/stable
/grove/yocto-layers/meta-blinx/+/fruit:a3f5d8e12b4c6789...
```

- The system SHALL support storage and retrieval of packages in multiple formats:
  - OpenWrt packages (opkg/ipk files)
  - Debian packages (dpkg/deb files)
  - Yocto layers and recipes
  - Root filesystem images (squashfs, ext4, etc.)
  - Kernel images and modules
  - Python packages (wheels, eggs, sdists)
  - Rust crates
  - C++ libraries (static/dynamic libraries, headers)
  - Generic binary artifacts
  - Container images (OCI format)
  - NPM packages
  - JAR files

- The system SHALL extract and store basic format-specific metadata where applicable (version, architecture, package type)
- The system SHALL provide package-type aware search capabilities (e.g., searching for "numpy" would identify it as a Python package based on file type)
- The system SHALL maintain package manifests that describe dependencies and requirements
- The system SHALL support architecture-specific variants (x86_64, arm64, armhf, etc.)

- The system SHALL support exporting entire groves or selected trees to portable media

- The system SHALL generate self-contained export bundles with all dependencies included
- The system SHALL provide cryptographic signatures for export bundles
- The system SHALL support incremental exports containing only changes since last export
- The system SHALL support re-seeding a new Orchard instance from export bundles
- The system SHALL maintain full metadata and history during export/import operations

```
# Export entire grove for air-gap transfer
$ orchard export grove blinx-core --output blinx-core-2024.1.tar
Exporting grove 'blinx-core'...
✓ Collecting 1,247 fruits (45.6 GB)
✓ Generating manifest
✓ Creating signature
✓ Written to blinx-core-2024.1.tar (45.6 GB)

# Import into air-gapped Orchard instance
$ orchard import blinx-core-2024.1.tar
Importing grove 'blinx-core'...
✓ Verifying signature
✓ Checking integrity
✓ Importing 1,247 fruits
✓ Updating references
Import complete!

# Export specific trees with dependencies
$ orchard export tree blinx-core/kernel blinx-core/rootfs --deps --output release-2024.1.tar
```

- The system SHALL guarantee bit-for-bit identical artifacts for the same fruit ID
- The system SHALL never require internet access once dependencies are harvested
- The system SHALL support offline operation with cached/local content
- The system SHALL provide deterministic dependency resolution
- The system SHALL maintain complete build environment specifications
- The system SHALL support hermetic builds using only Orchard-provided dependencies

`orchard`

The Orchard CLI (`orchard`) provides local dependency management and resolution for projects. Users can clone a repository and use the CLI to automatically fetch all required dependencies before building.

```
# Clone the Blinx custom Linux distribution project
$ git clone https://gitlab.company.com/embedded/blinx-platform.git
$ cd blinx-platform

# The repository contains an orchard.ensure file defining all dependencies
$ cat orchard.ensure
```

`orchard.ensure`

```
# orchard.ensure - Dependency manifest for Blinx custom Linux distribution
# This file specifies all components needed to build the Blinx platform

grove: blinx-core
platform: linux-amd64

# Core Yocto components
seeds:
  # Yocto build system
  - tree: yocto-poky
    version: 4.0.13
    path: yocto/poky

  # BitBake build tool
  - tree: bitbake
    version: 2.0.0
    path: yocto/bitbake

  # Blinx BSP layer
  - tree: meta-blinx-bsp
    version: 2024.1.0
    path: yocto/meta-blinx

grove: blinx-core
platform: arm64

seeds:
  # Pre-built kernel for Blinx
  - tree: linux-kernel-blinx
    version: 6.1.55-blinx
    path: kernel/

  # Base root filesystem
  - tree: blinx-rootfs-base
    version: 2024.1.0
    path: rootfs/

grove: components
platform: arm64

seeds:
  # RTC (Real Time Clock) driver package
  - tree: rtc-ds3231-opkg
    version: 1.2.5
    path: packages/rtc/

  # Network time protocol daemon
  - tree: chrony-opkg
    version: 4.3.0
    path: packages/ntp/
  # Hardware monitoring tools
  - tree: lm-sensors-opkg
    version: 3.6.0
    path: packages/hwmon/

grove: customer-layers
platform: generic

seeds:
  # Customer-specific customization layer
  - tree: meta-customer-xyz
    tag: release-2024.1
    path: yocto/meta-customer/

  # Customer application packages
  - tree: customer-apps-bundle
    version: 3.5.0
    path: packages/apps/

# Build environment configuration
environment:
  MACHINE: blinx-armv8
  DISTRO: blinx
  BB_NUMBER_THREADS: 8
  PARALLEL_MAKE: -j8
  YOCTO_ROOT: ${ORCHARD_ROOT}/yocto
  KERNEL_PATH: ${ORCHARD_ROOT}/kernel
  ROOTFS_BASE: ${ORCHARD_ROOT}/rootfs
```

```
# Initialize Orchard in the project (reads orchard.ensure file)
$ orchard init
Orchard initialized for project 'blinx-platform'

# Fetch and resolve all dependencies
$ orchard harvest
Resolving dependencies from orchard.ensure...
✓ Fetching yocto-poky @ 4.0.13 (892.3 MB)
✓ Fetching bitbake @ 2.0.0 (12.4 MB)
✓ Fetching meta-blinx-bsp @ 2024.1.0 (156.7 MB)
✓ Fetching linux-kernel-blinx @ 6.1.55-blinx (78.3 MB)
✓ Fetching blinx-rootfs-base @ 2024.1.0 (2.1 GB)
✓ Fetching rtc-ds3231-opkg @ 1.2.5 (124 KB)
✓ Fetching chrony-opkg @ 4.3.0 (1.8 MB)
✓ Fetching lm-sensors-opkg @ 3.6.0 (856 KB)
✓ Fetching meta-customer-xyz @ release-2024.1 (45.2 MB)
✓ Fetching customer-apps-bundle @ 3.5.0 (234.5 MB)
✓ Verifying checksums...
✓ Extracting to .orchard/...
All dependencies harvested successfully!
Build environment configured.

# Now build the Blinx distribution
```

```
$ eval $(orchard env export)  # Load environment variables
$ cd yocto/poky
$ bitbake blinx-image-full
```

```
# Basic Commands
orchard init                      # Initialize Orchard in current directory
orchard harvest                    # Fetch all dependencies from orchard.ensure
orchard plant <package> <version>  # Add a new dependency
orchard prune                      # Remove unused cached dependencies
orchard status                     # Show current dependency status

# Query Commands
orchard search <pattern>           # Search available packages (format-aware)
orchard info <package>             # Show package details and versions
orchard tree                       # Display dependency tree

# Publishing Commands
orchard cultivate <file> <package> # Upload new artifact
orchard tag <package> <fruit> <tag> # Add tag to existing fruit
orchard graft <src> <dest>         # Create alias

# Export/Import Commands (for air-gap)
orchard export grove <name>       # Export entire grove
orchard export tree <path>        # Export specific tree
orchard import <bundle>           # Import from bundle
orchard verify <bundle>           # Verify bundle integrity

# Advanced Commands
orchard lock                      # Generate orchard.lock file
orchard verify                    # Verify all checksums
orchard cache clean               # Clear local cache
orchard grove list                # List accessible groves
orchard mirror <remote> <local>   # Mirror remote grove locally

# Environment Commands
orchard env export                # Export environment variables for build system
orchard env show                  # Display environment variables
orchard env export --format <fmt> # Export in specific format (cmake, make, etc.)
```

The CLI SHALL resolve package dependencies from **orchard.ensure** files, which specify required packages and their versions.

- The system SHALL support YAML format for orchard.ensure files
- The system SHALL validate orchard.ensure files against a defined schema before processing
- The system SHALL provide clear error messages for malformed ensure files

- **grove**: The source grove (project) from which to fetch packages
  - Multiple grove sections MAY be specified in a single file
  - Each grove section MAY include a platform specifier
- **seeds**: Array of package dependencies to fetch
  - Each seed MUST specify a tree (package name)
  - Each seed MUST specify either a version, tag, or fruit ID
  - Each seed MUST specify a path for local extraction

Each seed entry SHALL support the following fields:

- **tree** (required): Name of the package within the grove
- **version** (optional): Semantic version string (e.g., "1.2.3", ">=1.0.0")
- **tag** (optional): Named tag reference (e.g., "latest", "stable", "release-2024.1")
- **fruit** (optional): Direct SHA256 hash for exact reproducibility
- **path** (required): Local filesystem path relative to `.orchard/` where content will be extracted
  - Paths MAY be arbitrarily nested (e.g., `path: tools/compilers/gcc/arm/v12/`)
  - The system SHALL create all necessary parent directories
  - The system SHALL preserve the extracted content's internal structure

- The system SHALL support platform-specific dependency resolution
- Platform specifiers MAY include:
  - Architecture: `linux-amd64`, `linux-arm64`, `darwin-amd64`, `windows-amd64`
  - Generic: `generic` for platform-independent packages
  - Custom: User-defined platform strings for specialized environments
- If no platform is specified, `generic` SHALL be assumed

- The system SHALL support an `environment` section for defining environment variables in the ensure file
- Environment variables MAY reference `${ORCHARD_ROOT}` for portable paths
- The system SHALL NOT automatically set environment variables
- The system SHALL provide an `orchard env` command to export environment variables for build system use
- Environment variable definitions SHALL support:
  - Simple values: `KEY: value`
  - Path construction: `PATH: ${ORCHARD_ROOT}/bin:${PATH}`
  - Reference to seed paths: `TOOL_HOME: ${ORCHARD_ROOT}/tools/gcc`

```
# Export environment variables to stdout (for evaluation)
$ orchard env export
export YOCTO_ROOT="/home/user/blinx/.orchard/tools/yocto"
export KERNEL_PATH="/home/user/blinx/.orchard/kernel"
export PATH="/home/user/blinx/.orchard/tools/bin:${PATH}"

# Source directly into current shell
$ eval $(orchard env export)

# Generate a shell script file
$ orchard env export --output orchard-env.sh
$ source orchard-env.sh
```

```
# Show environment variables without export prefix (for debugging)
$ orchard env show
YOCTO_ROOT=/home/user/blinx/.orchard/tools/yocto
KERNEL_PATH=/home/user/blinx/.orchard/kernel
PATH=/home/user/blinx/.orchard/tools/bin:${PATH}

# Export in different formats for various build systems
$ orchard env export --format cmake > OrchardPaths.cmake
$ orchard env export --format make > orchard.mk
```

- The system SHALL support conditional seeds based on:
    - Platform/architecture: `when: platform == "linux-arm64"`
    - Environment variables: `when: $BUILD_TYPE == "debug"`
    - Feature flags: `when: features.contains("rtc-support")`

```
# orchard.ensure schema version (for future compatibility)
version: 1.0

# Global settings
settings:
  cache_dir: .orchard/cache
  parallel_downloads: 4
  verify_signatures: true

# Package sources
grove: blinx-core
platform: linux-arm64

seeds:
  - tree: yocto-poky
    version: 4.0.13
    path: tools/yocto/poky

  - tree: linux-kernel
    tag: stable-6.1
    path: kernel/
    when: $KERNEL_BUILD == "true"

grove: third-party
platform: generic

seeds:
  - tree: documentation
    version: ">=2.0.0 <3.0.0"  # Version range
    path: docs/

# Direct fruit reference for exact reproducibility
grove: frozen-deps
```

```yaml
seeds:
  - tree: critical-lib
    fruit: a3f5d8e12b4c67890abcdef1234567890abcdef1234567890abcdef12345678
    path: libs/critical/

# Environment setup
environment:
  YOCTO_ROOT: ${ORCHARD_ROOT}/tools/yocto
  KERNEL_PATH: ${ORCHARD_ROOT}/kernel
  PATH: ${ORCHARD_ROOT}/tools/bin:${PATH}
  LD_LIBRARY_PATH: ${ORCHARD_ROOT}/libs:${LD_LIBRARY_PATH}

# Feature flags (optional)
features:
  - rtc-support
  - debug-symbols
```

- The system SHALL extract package contents to the specified path under `.orchard/`
- If the package is an archive (tar, zip, etc.), it SHALL be extracted
- If the package is a single file, it SHALL be placed at the specified path
- The system SHALL preserve file permissions and attributes during extraction
- The system SHALL handle path conflicts according to configurable policy (overwrite, skip, error)

After resolving dependencies, Orchard generates an `orchard.lock` file to ensure reproducible builds:

```yaml
# orchard.lock - Generated by Orchard. DO NOT EDIT MANUALLY.
# This file locks specific fruit IDs for reproducible builds.

timestamp: 2024-01-15T10:30:00Z
orchard_version: 1.0.0

resolved:
  - grove: blinx-core
    tree: yocto-poky
    version: 4.0.13
    fruit: a3f5d8e12b4c67890abcdef1234567890abcdef1234567890abcdef12345678
    size: 935059456

  - grove: blinx-core
    tree: linux-kernel-blinx
    version: 6.1.55-blinx
    fruit: b4f6e9f23c5d78901bcdef2345678901bcdef2345678901bcdef2345678901b
    size: 82063360

  - grove: components
    tree: rtc-ds3231-opkg
    version: 1.2.5
    fruit: c5g7f0g34d6e89012cdef3456789012cdef3456789012cdef3456789012cde
    size: 126976
```

```
# ... additional resolved dependencies
```

- The CLI SHALL support manifest files (orchard.ensure) that specify required packages and versions
- The CLI SHALL support environment-specific dependency resolution (OS, architecture)
- The CLI SHALL support lock files (orchard.lock) to ensure reproducible builds
- The CLI SHALL validate dependency compatibility before download
- The CLI SHALL support conditional dependencies based on platform or build configuration

- The system SHALL support configurable access control at the grove (project) level
- The system SHALL provide three permission levels: read, write, and admin
- The system SHALL support both public and private groves
- Groves SHALL be public by default unless explicitly configured as private
- The system SHALL support both local authentication and federated authentication (OIDC/SAML)
- The system SHALL provide API key authentication for programmatic access

- The system SHALL support revoking access to trees without deleting the underlying fruit
- The system SHALL maintain access history for audit purposes
- The system SHALL support time-limited access tokens for temporary access

- The system SHALL support creating grafts (aliases) that point to specific fruits
- The system SHALL allow multiple grafts to point to the same content
- The system SHALL support graft chains (graft → graft → fruit)
- The system SHALL maintain graft history for rollback capability
- The system SHALL support atomic graft updates to prevent inconsistent states

- The system SHALL support semantic versioning for packages
- The system SHALL resolve version ranges to specific fruits
- The system SHALL support version pinning for reproducible builds
- The system SHALL track version lineage and relationships

- The system SHALL track which projects consume specific packages
- The system SHALL maintain a dependency graph of package relationships
- The system SHALL identify consumers through API access patterns
- The system SHALL support explicit consumer registration

- The system SHALL identify consumers when new package versions are published
- The system SHALL support automated merge/pull request creation for consumer updates
- The system SHALL provide webhooks for harvest completion notifications
- The system SHALL support configurable update policies (automatic, manual approval, blocked)
- The system SHALL generate compatibility reports for proposed updates
- The system SHALL support rollback notifications if updates cause failures

```
1. Developer publishes new RTC driver package (rtc-ds3231-opkg v1.3.0) to Orchard
2. Orchard identifies 8 Blinx-based projects consuming the RTC driver
3. For each consumer with auto-update enabled:
   - Orchard creates a branch updating orchard.ensure
   - Runs compatibility checks against Blinx kernel version
   - Opens MR with title: "🌳 Orchard: Update rtc-ds3231-opkg to v1.3.0"
   - Includes changelog and kernel compatibility notes in MR description
4. CI/CD runs on the MR to validate the update
5. If tests pass, MR can be auto-merged based on policy
```

- The system SHALL analyze update impact before proposing changes
- The system SHALL support update scheduling and batching
- The system SHALL maintain update history and success/failure metrics
- The system SHALL provide breaking change detection and warnings

- The system SHALL verify SHA256 hash on every upload and download
- The system SHALL support additional checksum algorithms (MD5, SHA1) for compatibility
- The system SHALL detect and report corruption immediately
- The system SHALL maintain checksums in S3 object metadata

- The system SHALL maintain immutable audit logs of all operations
- The system SHALL track complete provenance for all artifacts
- The system SHALL support compliance reporting (who uploaded what, when, from where)
- The system SHALL provide content scanning integration points for security tools

- The system SHALL use S3 multipart uploads for files larger than 100MB
- The system SHALL implement S3 request parallelization for performance
- The system SHALL use S3 Transfer Acceleration when available
- The system SHALL support S3-compatible CDN integration for global distribution
- The system SHALL implement intelligent caching of frequently accessed fruits

- The system SHALL support high-throughput parallel uploads and downloads

- The system SHALL provide resumable uploads for large files
- The system SHALL support partial content requests (HTTP range requests)
- The system SHALL provide bulk operations for efficiency
- The system SHALL implement connection pooling for S3 operations

- The system SHALL provide plugins for common CI/CD systems (Jenkins, GitLab CI, GitHub Actions)
- The system SHALL support build artifact promotion workflows
- The system SHALL integrate with existing package managers (opkg, apt, pip, npm, cargo, etc.)
- The system SHALL provide build reproducibility verification

```yaml
# .gitlab-ci.yml
stages:
  - fetch
  - build
  - package
  - publish

variables:
  ORCHARD_GROVE: "blinx-core"

fetch-dependencies:
  stage: fetch
  image: orchard/cli:latest
  script:
    - orchard init
    - orchard harvest
  artifacts:
    paths:
      - .orchard/
    expire_in: 1 day

build-blinx:
  stage: build
  image: crops/poky:latest
  dependencies:
    - fetch-dependencies
  script:
    - eval $(orchard env export)  # Load Orchard environment variables
    - cd yocto/poky
    - source oe-init-build-env
    - bitbake blinx-image-full
  artifacts:
    paths:
      - build/tmp/deploy/images/
    expire_in: 1 week

package-rootfs:
  stage: package
  image: orchard/cli:latest
  dependencies:
    - build-blinx
  script:
    - cd build/tmp/deploy/images/blinx-armv8/
    - tar czf blinx-rootfs-${CI_COMMIT_TAG}.tar.gz blinx-image-full-*.rootfs.ext4
  artifacts:
    paths:
      - build/tmp/deploy/images/blinx-armv8/*.tar.gz

publish-to-orchard:
  stage: publish
  image: orchard/cli:latest
  only:
    - tags
  dependencies:
    - package-rootfs
  script:
    - |
      orchard cultivate \
        build/tmp/deploy/images/blinx-armv8/blinx-rootfs-${CI_COMMIT_TAG}.tar.gz \
        blinx-rootfs
    - orchard tag blinx-core/blinx-rootfs latest ${CI_COMMIT_TAG}
    - orchard tag blinx-core/blinx-rootfs stable ${CI_COMMIT_TAG}
```

- The system SHALL expose metrics for storage usage, deduplication rates, and access patterns
- The system SHALL provide real-time alerting for system issues
- The system SHALL support distributed tracing for request debugging
- The system SHALL maintain SLA metrics for availability and performance
- The system SHALL integrate with S3 CloudWatch metrics when using AWS

Orchard Term	Traditional Term	Description
Grove	Project	Top-level organizational unit
Tree	Package	Named collection of related artifacts
Fruit	Instance	Specific content identified by SHA256
Seed	Dependency	Required package specification
Harvest	Download/Fetch	Retrieve dependencies
Cultivate	Upload/Publish	Store new artifact
Graft	Alias	Alternative name for content
Prune	Clean/GC	Remove unused local cache

Pollination	Consumer Update	Cross-project dependency updates
Orchard	System Name	The overall platform

- We need an example thread for how this will work and what pieces would be included

Legend:

- [ ] Desired future state
- [x] Current state

- [ ] With a Windows 11 laptop, BEN access, BSF access, and BSFC, I have everything I need to start using Orchard
- [ ] I set up my right to HTTPS clone in gitlab using credential manager in a shell prescribed by the BSF team
- [ ] In a shell, I git clone via HTTPS the "origin of the universe" repo for Boeing Linux, which contains the Orchard configs. This is similar to a collection of google-repo manifests in a git repo. The repo contains directories for various product configs. Each directory has a "default" config that tracks the latest branches. In addition, each directory has many "snapshots" of orchard configs such as my_project_0.2.3.orchard. These snapshots have no branch references, and contain only shas or tags that have been configured to be immutable.
- [ ] I cd into the repo and "orchard init my_project_default.orchard" or init with whatever config I want to work on.
- [ ] Orchard invokes whatever tools are needed to set up a Boeing Linux working environment, such as git clone for a program repo, matching dev-container docker pulling, etc.
- [x] I invoke the Boeing Linux native tool to clone yocto dependencies and do builds (kas and yocto). Everything not considered "part of the environment" or dev-container is cloned now. From this command down, the Boeing Linux build is reproduceable from kas config yaml. If both the environment (Orchard) and the post kas/yocto fetch were archived, I would expect to be able to reproduce this build forever. Note that the kas yaml uses HTTPS clones with an explicit token passed in provided by the environment, but that is not ideal.
- [x] I issue kas/yocto build commands which fetch using HTTPS through artifactory for anything in SRC_URI. Similar to above, I think we need some improvements here for authentication consistency, but none of those changes touch Orchard.
- [x] My changes eventually hit an MR and get merged to a release branch.
- [x] After a successful merge pipeline (which includes some automated testing), I "push the button" to tag and promote that pipeline's artifacts to GPR as a release.
- [x] I notify the customer as per our release process and they can pull from GPR and do whatever they do after that.

- [ ] I am aware of the notion of the "Image Composition Toolkit", which admittedly does not exist
- [ ] I expect to use the Image Composition Toolkit with my apps to compose a final rootfs in the desired format. We have multiple customers who are layering squashfs for this purpose, but I'll ignore that for now.
- [ ] I run a pipeline that pulls from the Boeing Linux GPR a specific semver and adds my apps on top using the Image Composition Toolkit

- [ ] I am aware of the notion of Orchard
- [ ] I create an Orchard config in a program repo that pulls from the Boeing Linux GPR for my program by semver, and also pulls from other Fabric program's GPRs, as well as my Program GPRs
- [ ] I run Orchard locally or in CI to compose my "final" rootfs. I perform additional steps as needed to create LSAPs and number my software

parts, ?where everything else up to this point is just by semver?

Comments