The Install Architectural Style

A Comprehensive Reference for Mapping Software to File Systems

Contents

1 Overview

The install style describes the mapping of components in the software architecture to a file system in the production environment. This allocation view documents how software artifacts—executables, libraries, configuration files, data files, and other resources—are organized within the directory structure of the target system.

While the deployment style addresses where software runs (which computing nodes), the install style addresses how software is organized on those nodes (which files and directories). Together, these allocation views provide a complete picture of the physical manifestation of software architecture.

The install style is essential for understanding system organization, managing software distribution, controlling access and permissions, maintaining configuration integrity, and enabling efficient operations. It bridges the gap between abstract software components and the concrete file system artifacts that constitute the installed system.

1.1 Scope and Applicability

The install style applies to any software system that must be installed on a file system before execution. This includes traditional server applications with complex directory structures, desktop applications distributed to end-user machines, mobile applications packaged for app stores, embedded systems with constrained file system resources, web applications deployed to application servers, and containerized applications with layered file systems.

The style is particularly valuable when the system exhibits significant file organization complexity, must conform to platform-specific conventions, requires careful management of configuration and data separation, has security requirements affecting file permissions, must support multiple installation scenarios or configurations, or needs to integrate with package management systems.

1.2 Relationship to Other Architectural Views

Install views complement other architectural perspectives. Module views describe code organization during development; install views describe artifact organization during deployment. Component-and-connector views describe runtime behavior; install views describe the static file structure that enables that behavior. Deployment views describe allocation to computing nodes; install views describe allocation within those nodes.

A complete allocation perspective typically includes both deployment views (showing which nodes host which components) and install views (showing how those components are organized on each node's file system).

1.3 Historical Context

The install style has evolved significantly with changing software distribution paradigms. Early systems had simple, monolithic installations where all files resided in a single directory. Operating system conventions emerged (such as Unix's /usr hierarchy and Windows' Program Files), establishing standard locations for different artifact types. Package managers introduced dependency tracking and standardized installation procedures. Modern containerization has created immutable, layered file systems that fundamentally change installation assumptions.

Understanding this evolution helps architects make appropriate decisions for their target environments and distribution mechanisms.

2 Elements

The install style comprises two fundamental categories of elements: software elements that must be installed and environmental elements that constitute the file system structure.

2.1 Software Elements

Software elements in the install style are the artifacts derived from component-and-connector components that must be placed in the file system. These represent the tangible outputs of the build process that constitute the installed system.

2.1.1 Types of Software Elements

Software elements encompass several categories. Executable files include compiled binaries, interpreted scripts, and bytecode files that can be directly executed or interpreted by a runtime environment. These are the primary entry points for system functionality.

Library files include shared libraries (DLLs on Windows, .so files on Unix, .dylib on macOS), static libraries, and managed assemblies that provide reusable functionality to executables.

Configuration files include settings files (in formats such as XML, JSON, YAML, INI, or properties files), environment-specific configurations, feature flags, and runtime tuning parameters that control system behavior without code changes.

Data files include static data required for operation, such as lookup tables, reference data, templates, localization resources, and machine learning models.

Resource files include assets such as images, icons, fonts, stylesheets, JavaScript bundles, and other non-executable content required by the application.

Documentation files include user manuals, API documentation, license files, release notes, and README files distributed with the software.

Log and temporary file placeholders include designated locations for runtime-generated files such as logs, caches, temporary files, and working data.

Plugin and extension files include optional components that extend base functionality, typically loaded dynamically based on presence in designated directories.

2.1.2 Essential Properties of Software Elements

When documenting software elements for installation, architects should capture several categories of properties.

File identity properties include the canonical file name, file extension indicating type, version identifier (often embedded in name or metadata), and digital signature or checksum for integrity verification.

Size and resource properties encompass file size (compressed and uncompressed), memory footprint when loaded, and disk space requirements including growth projections for data files.

Dependency properties describe other software elements required (shared libraries, frameworks, runtimes), external system dependencies (database clients, system libraries), and load-order dependencies (files that must be present before this element loads).

Permission properties specify required file system permissions (read, write, execute), ownership requirements (user, group), and special attributes (setuid, setgid, immutable flags).

Platform properties identify target operating systems and versions, processor architecture requirements (x86, x64, ARM), and runtime environment requirements (JVM version, .NET version, Python interpreter).

Lifecycle properties describe whether the file is static (unchanged after installation) or dynamic (modified during operation), update frequency and mechanism, and backup and recovery requirements.

Origin properties trace source component in C&C architecture, build artifact producing this element, and version control location of source.

2.2 Environmental Elements

Environmental elements represent the file system structure that hosts software elements. These are configuration items—files and directories—that constitute the installation target.

2.2.1 Types of Environmental Elements

Directory elements are containers that organize other environmental elements hierarchically. Directories may be system-defined (standard operating system locations), application-specific (created during installation), or user-specific (per-user configuration or data).

File system roots are the top-level mount points or drive letters that anchor directory hierarchies. On Unix systems, this is typically the root directory; on Windows, these are drive letters.

Symbolic links and junctions are file system references that point to other locations, enabling flexible organization and compatibility across different installation scenarios.

Mount points are locations where additional file systems are attached, relevant for systems spanning multiple storage devices or network file systems.

2.2.2 Essential Properties of Environmental Elements

Location properties specify absolute path within the file system hierarchy, relative path from installation root, and environment variables or registry keys that reference this location.

Capacity properties include available disk space, inode availability (on Unix systems), and quota limitations that may apply.

Permission properties describe access control lists or permission bits, ownership (user and group on Unix, security descriptors on Windows), and inheritance rules for contained elements.

Platform conventions indicate whether the location follows platform-specific conventions (such as Filesystem Hierarchy Standard on Linux, Windows conventions, or macOS application bundles) and which convention version applies.

Volatility properties indicate whether contents persist across reboots, whether the location is backed up, and retention policies that apply.

Sharing properties describe whether the location is shared across users, shared across applications, or private to this installation.

3 Relations

Relations in the install style define how software elements map to environmental elements and how environmental elements relate to each other.

3.1 Allocated-To Relation

The primary relation is *allocated-to*, specifying that a software element resides at a particular location in the file system. This relation answers the fundamental question: where is this artifact installed?

3.1.1 Properties of Allocated-To

Installation timing indicates when the allocation occurs. Pre-installation means the element exists before the main installation process (prerequisites). Installation-time means the element is placed during the standard installation procedure. Post-installation means the element is created after installation (first-run setup, user registration). Runtime means the element is created during normal operation (caches, logs).

Conditionality describes whether allocation always occurs or depends on installation options. Mandatory elements are always installed. Optional elements are installed based on user selection or detected requirements. Conditional elements are installed only when specific conditions are met (platform detection, dependency satisfaction).

Mutability indicates whether the installed element may change. Immutable elements are never modified after installation. Configuration-mutable elements may be modified by administrators or users to customize behavior. System-mutable elements are modified by the system during operation (logs, caches). Update-mutable elements are modified only during software updates.

Source specifies where the element originates during installation—from the installation package, downloaded during installation, generated during installation, or copied from an existing location.

3.2 Containment Relation

The *containment* relation specifies that one environmental element (directory) contains another environmental element (file or subdirectory). This relation defines the hierarchical structure of the file system.

3.2.1 Properties of Containment

Depth indicates the nesting level within the containment hierarchy, relevant for path length limitations and organizational clarity.

Exclusivity indicates whether the container holds only elements from this installation (dedicated directory) or may contain elements from multiple sources (shared directory).

Creation responsibility specifies whether the container is created by the installation process, expected to pre-exist, or created on demand when first needed.

3.3 Depends-On Relation

The *depends-on* relation indicates that one software element requires another to be present and accessible. This relation is critical for installation ordering and validation.

3.3.1 Properties of Depends-On

Dependency type distinguishes build-time dependencies (needed to create the element), install-time dependencies (needed during installation), load-time dependencies (needed when the element loads), and runtime dependencies (needed during execution).

Resolution mechanism describes how dependencies are located—fixed path, search path, configuration setting, or dynamic discovery.

Version constraints specify acceptable versions of the dependency, including minimum version, maximum version, or exact version requirements.

Optionality indicates whether the dependency is required (installation fails without it), recommended (installation succeeds but functionality is limited), or optional (enhanced functionality if present).

3.4 References Relation

The *references* relation indicates that one software element refers to another by path or identifier. Unlike depends-on, references may be to elements outside the installation.

3.4.1 Properties of References

Reference type describes the nature of the reference—file path (absolute or relative), URL, registry key, environment variable, or symbolic name.

Resolution time indicates when the reference is resolved—installation time, load time, or runtime.

Fallback behavior describes what happens if the referenced element is not found—error, default value, or alternative location.

4 Constraints

The install style operates within constraints imposed by file systems, operating system conventions, and organizational policies.

4.1 Hierarchical Structure Constraint

Files and folders are organized in a tree structure following an *is-contained-in* relation. Each file or directory (except the root) has exactly one parent directory. This fundamental constraint shapes all installation decisions.

Implications include the requirement that paths must be valid within the hierarchy, the limitation that circular containment is impossible, and the consideration that path length limitations may apply (260 characters on older Windows systems, longer limits on modern systems and Unix).

4.2 Naming Constraints

File and directory names must conform to file system rules. Character restrictions prohibit certain characters in names (such as /, \, :, *, ?, ”, ¡, ¿, — on Windows). Length limits bound individual name segment length and total path length. Case sensitivity varies by file system (case-sensitive on most Unix systems, case-insensitive on Windows and macOS by default). Reserved names prohibit certain names on some platforms (CON, PRN, AUX, NUL on Windows).

4.3 Permission Constraints

Installation must respect and establish appropriate permissions. Installation privileges may require administrative or root access for system locations. Runtime permissions must allow appropriate access for the executing user. Security policies may restrict installation to approved locations or require specific ownership.

4.4 Platform Convention Constraints

Operating systems establish conventions for file organization that installations should follow.

On Unix and Linux systems, the Filesystem Hierarchy Standard (FHS) defines standard directories: /usr for read-only program files, /etc for configuration, /var for variable data, /opt for optional software packages, and user home directories for per-user data.

On Windows systems, conventions include Program Files for application executables, ProgramData for shared application data, user AppData directories for per-user data (Roaming for synchronized data, Local for machine-specific data), and the Windows directory for system files.

On macOS systems, conventions include /Applications for GUI applications, /Library for system-wide resources, user Library directories for per-user resources, and application bundles (.app directories) containing self-contained applications.

4.5 Isolation Constraints

Modern systems often impose isolation requirements. Container file systems create layered, isolated views of the file system. Sandboxing restricts application access to designated directories. App store requirements mandate specific installation structures and signing.

4.6 Space Constraints

File system capacity limits installation options. Disk quotas may limit space available to specific users or groups. Partition boundaries may require distributing installation across mount points. Reserved space for system operation must be preserved.

5 What the Style is For

The install style serves several critical purposes throughout the software lifecycle.

5.1 Installation Planning

Install views enable teams to plan the installation process by defining directory structure requirements before deployment, identifying permissions and privileges needed, planning installation scripts and procedures, and anticipating platform-specific variations.

Effective installation planning requires understanding target environment conventions, organizational policies for software installation, integration requirements with existing systems, and rollback and recovery procedures.

5.2 Configuration Management

Install views support configuration management by documenting the expected state of installed systems, enabling verification that installations match specifications, supporting detection of unauthorized modifications, and facilitating consistent reproduction of installations.

Configuration management tools (Ansible, Puppet, Chef, Salt) use install specifications to ensure systems remain in desired states.

5.3 Security Analysis

The physical organization of files affects security posture. Install views enable security teams to analyze file permissions and ownership, identify sensitive files requiring protection, plan access control strategies, detect potential security misconfigurations, and ensure compliance with security policies.

Security-relevant installation decisions include separating executables from data to prevent code injection, protecting configuration files containing credentials, ensuring appropriate permissions on log files, and isolating user-modifiable content from system files.

5.4 Troubleshooting and Support

When systems malfunction, install views guide diagnosis by documenting expected file locations for investigation, identifying configuration files affecting behavior, locating log files for error analysis, and enabling comparison between expected and actual installation state.

Support teams rely on install documentation to guide users through configuration changes, identify common installation problems, and develop remediation procedures.

5.5 Upgrade and Migration Planning

Install views inform upgrade strategies by identifying files that must be preserved during upgrades, documenting migration paths for configuration and data, enabling rollback by preserving previous versions, and planning for breaking changes in file organization.

Upgrade planning must consider backward compatibility of file locations, data migration procedures, configuration format changes, and cleanup of obsolete files.

5.6 Compliance and Auditing

Regulatory requirements often mandate specific controls over installed software. Install views support compliance by documenting the authorized installation state, enabling audit verification of actual installations, supporting change tracking for regulated systems, and demonstrating separation of duties in installation procedures.

5.7 Packaging and Distribution

Install views guide packaging decisions by defining package contents and structure, specifying pre- and post-installation scripts, declaring dependencies on other packages, and enabling reproducible installation across environments.

Different distribution mechanisms (operating system packages, container images, application installers) require different approaches to expressing install specifications.

6 Notations

Install views can be represented using various notations suited to different audiences and purposes.

6.1 Directory Tree Diagrams

The most intuitive notation shows the file system hierarchy as a tree structure, using indentation or connecting lines to show containment relationships. This notation is easy to understand and create, suitable for documentation and communication. However, it can become unwieldy for large installations and has limited ability to show properties.

Example representation using text-based tree notation:

```
/opt/myapp/
+-- bin/
|   +-- myapp
|   \-- myapp-cli
+-- lib/
|   +-- libcore.so
|   \-- plugins/
|       \-- plugin-auth.so
+-- etc/
|   +-- myapp.conf
|   \-- logging.xml
+-- var/
|   +-- log/
|   \-- cache/
\-- share/
    \-- doc/
        \-- README.md
```

6.2 Tabular Notation

Tables can document installation mappings with associated properties, providing structured documentation with consistent property capture, suitability for detailed specification, and ease of automated processing. Limitations include difficulty showing hierarchical relationships clearly and verbosity for large installations.

Table columns typically include path (absolute or relative file path), type (directory, executable, configuration, data, etc.), permissions (file system permissions), owner (user and group ownership), source (origin in installation package), and notes (additional information).

6.3 UML Component Diagrams

UML artifacts and deployment diagrams can represent installation structure. Artifacts represent installed files; nodes can represent directories. This notation provides integration with other UML views and formal semantics, though it may be overkill for simple installations and requires UML familiarity.

6.4 Package Manager Specifications

Many systems express installation structure through package manager formats. RPM spec files define Linux RPM package contents. Debian control files define .deb package structure. Windows Installer XML (WiX) defines Windows installer content. Dockerfile instructions build container images with specific file structures.

These specifications serve as executable documentation, directly driving the installation process while documenting the intended structure.

6.5 Configuration Management Declarations

Infrastructure-as-code tools express desired installation state. Ansible playbooks declare files and directories to create. Puppet manifests specify file resources. Chef recipes define file system state.

These declarations combine documentation and automation, ensuring that documented structure matches actual installations.

6.6 Formal Architecture Description Languages

For rigorous analysis, formal ADLs can represent installation structure. AADL supports hardware and software allocation modeling. SysML can represent physical system organization.

7 Quality Attributes

Installation decisions directly affect several quality attributes of the resulting system.

7.1 Security

File organization significantly impacts security posture. Permission structure matters because proper permissions prevent unauthorized access and modification; the principle of least privilege should guide permission decisions. Separation of concerns is achieved through isolating executables, configuration, and data to limit impact of compromises. Sensitive data protection requires configuration files with credentials to have restricted permissions. Audit capability means predictable file locations enabling security monitoring and intrusion detection.

Poor installation decisions can create vulnerabilities: world-writable directories enabling code injection, sensitive files readable by unprivileged users, executables in user-writable locations enabling privilege escalation, and predictable temporary file locations enabling race conditions.

7.2 Modifiability

Installation structure affects ease of system modification. Configuration externalization means separating configuration from code enabling runtime modification. Plugin architecture support allows directory structures supporting dynamic extension loading. Update isolation means organizing files to enable incremental updates without full reinstallation. Version coexistence means supporting multiple versions simultaneously when needed.

Well-organized installations support modification by clearly separating static and dynamic content, providing extension points for customization, enabling configuration changes without reinstallation, and supporting feature toggling through configuration.

7.3 Portability

Installation choices affect cross-platform deployment. Path conventions mean using relative paths and avoiding hardcoded absolute paths. Platform abstraction means accommodating different platform conventions through configuration. Containerization means immutable, self-contained installations providing maximum portability.

Portable installations avoid platform-specific path separators in code, hardcoded paths to system directories, assumptions about case sensitivity, and reliance on platform-specific features without abstraction.

7.4 Performance

File organization can impact performance. Locality means placing frequently accessed files together to improve cache behavior. File system selection means matching file system features to access patterns. Compression means balancing storage efficiency against decompression overhead. Loading efficiency means organizing files to minimize load-time I/O operations.

Performance-sensitive installations consider solid-state versus spinning disk characteristics, network file system latency, memory-mapped file efficiency, and startup time optimization through file organization.

7.5 Reliability

Installation structure affects system reliability. Integrity verification means supporting checksums and signatures for installed files. Corruption isolation means organizing files so corruption in one area doesn't affect others. Recovery support means enabling restoration from backups through clear separation of state. Graceful degradation means handling missing optional components appropriately.

7.6 Usability

For end-user installations, organization affects usability. Discoverability means users should find configuration and data files where expected. Convention compliance means following platform conventions that users understand. Documentation accessibility means placing documentation where users will find it. Uninstallation cleanliness means enabling complete removal without leaving orphaned files.

8 Common Installation Patterns

Several recurring patterns address common installation challenges.

8.1 Single-Directory Installation

All application files reside in a single directory subtree, providing complete self-containment. Benefits include simple deployment (copy entire directory), easy uninstallation (delete entire directory), clear boundaries (everything application-related in one place), and portability (relocatable without path changes).

This pattern is appropriate for portable applications not requiring system integration, development and testing environments, applications distributed as archives (zip, tar), and situations where administrative privileges are unavailable.

Limitations include inability to integrate with system services, potential duplication of shared libraries, and conflict with platform conventions expecting distributed organization.

8.2 Distributed Installation

Application files are distributed across standard system directories according to platform conventions. Executables go in standard binary directories, libraries in library directories, configuration in configuration directories, and data in data directories.

Benefits include platform convention compliance, integration with system tools (package managers, service managers), shared library efficiency, and standard locations for administrators.

This pattern is appropriate for system services and daemons, applications installed via package managers, software requiring administrative privileges, and production server deployments.

Limitations include complex installation and uninstallation, platform-specific organization requirements, and potential conflicts with other applications.

8.3 User-Space Installation

Applications install entirely within user directories, requiring no administrative privileges. Benefits include no administrator required, per-user customization enabled, isolation from other users, and reduced system-wide impact.

This pattern is appropriate for desktop applications on shared systems, development tools, applications installed via language-specific package managers (pip, npm), and situations where users lack administrative access.

Limitations include duplication across users, inability to provide system services, and potential inconsistency between user installations.

8.4 Layered Installation

Installation builds up through layers, each adding or modifying files. This is the fundamental pattern for container images. Benefits include efficient storage through layer sharing, clear provenance of each file, immutable base layers with mutable upper layers, and reproducible builds through layer caching.

This pattern is appropriate for containerized applications (Docker, OCI), immutable infrastructure approaches, applications requiring reproducible environments, and microservices deployments.

Limitations include layer size optimization challenges, complexity of layer management, and potential for layer proliferation.

8.5 Side-by-Side Installation

Multiple versions of an application coexist in parallel directories. Benefits include safe upgrades (old version remains available), rollback capability, version-specific configuration, and gradual migration between versions.

This pattern is appropriate for applications requiring zero-downtime upgrades, systems where different components need different versions, testing new versions alongside production, and long-term support scenarios.

Limitations include increased disk space usage, complexity of version management, potential for configuration drift between versions, and dependency conflicts between versions.

8.6 Configuration Hierarchy Pattern

Configuration files are organized hierarchically with inheritance and override capabilities. Default configuration ships with the application. System configuration overrides defaults for all users. User configuration overrides system configuration for individual users. Instance configuration overrides for specific execution contexts.

Benefits include sensible defaults with customization capability, clear precedence rules, separation of shipped and local configuration, and support for multiple deployment contexts.

Implementation considerations include merge strategies (replace versus merge configuration sections), format compatibility across hierarchy levels, and documentation of override points.

8.7 Plugin Directory Pattern

A designated directory holds optional extension components loaded dynamically. The application scans the plugin directory at startup or runtime, loading and integrating discovered plugins.

Benefits include extensibility without core modification, optional functionality based on presence, third-party extension support, and reduced core application size.

Implementation considerations include plugin discovery mechanism, version compatibility checking, dependency handling among plugins, and security implications of dynamic loading.

9 Documentation Guidelines

Effective install documentation enables stakeholders to understand, create, and maintain installations.

9.1 Essential Content

Install documentation should include a directory structure map providing comprehensive documentation of all directories and their purposes. A file catalog lists all files installed with their type, purpose, and properties. Permission specifications detail required permissions for each file and directory. Platform variations note any differences across supported platforms. Dependency documentation captures external dependencies and how they are satisfied. Configuration guide explains configuration file purposes and customization options.

9.2 Views for Different Stakeholders

Different audiences need different perspectives on installation. Administrator view focuses on installation procedures, configuration options, and troubleshooting. Developer view emphasizes build artifact mapping, debug symbol locations, and development environment setup. Security view highlights permission requirements, sensitive file locations, and hardening options. Operations view covers monitoring integration, log locations, and maintenance procedures.

9.3 Maintaining Documentation Currency

Installation structures change over time. Strategies for keeping documentation current include generating documentation from package specifications, automated verification that documentation matches actual installations, documentation updates as part of change management procedures, and version-controlled documentation alongside installation scripts.

9.4 Common Documentation Pitfalls

Documentation quality suffers from incomplete coverage where undocumented files and directories create confusion. Stale documentation that does not reflect current installation structure misleads users. Missing rationale that omits explanation of why the structure was chosen hinders evolution. Platform assumptions that assume single-platform deployment without noting variations cause problems. Permission omission where failing to document required permissions leads to installation failures or security issues.

10 Relationship to DevOps Practices

Modern DevOps practices fundamentally influence installation architecture.

10.1 Infrastructure as Code

Installation specifications should be expressed as executable code. Benefits include version control of installation definitions, reproducible installations across environments, testable installation procedures, and documentation that cannot drift from reality.

Tools and approaches include package specifications (RPM spec, Debian control) defining operating system packages. Container definitions (Dockerfile, Buildah) create container images with specific file structures. Configuration management (Ansible, Puppet, Chef) declares desired file system state. Image builders (Packer, cloud-init) create machine images with pre-installed software.

10.2 Continuous Integration and Deployment

CI/CD pipelines automate installation artifact creation and deployment. Build stage creates installable artifacts from source. Package stage wraps artifacts in distributable format. Test stage verifies installation correctness. Deploy stage performs installation in target environments.

Installation architecture should support CI/CD by producing consistent artifacts from the same source, enabling automated installation without manual intervention, supporting installation verification and testing, and providing rollback capability for failed deployments.

10.3 Immutable Infrastructure

The immutable infrastructure pattern treats installed systems as disposable, replacing rather than modifying them. Implications for installation include installations being fully automated and reproducible, in-place updates being replaced by redeployment, configuration being externalized from installed images, and state being stored outside the installed system.

Container-based and serverless deployments exemplify immutable infrastructure, where installation creates an image that is never modified after creation.

10.4 GitOps

GitOps extends infrastructure as code by making Git the source of truth for installation state. Desired state is declared in Git repositories. Automated systems detect and reconcile drift. Changes flow through pull requests with review and approval.

Installation architecture supports GitOps through declarative installation specifications, automated application of declared state, drift detection and remediation, and audit trail of all installation changes.

10.5 Observability Integration

Modern installations must support observability requirements. Log aggregation support means installing log files in locations accessible to log collection agents. Metrics exposure means exposing installation health metrics. Tracing integration means supporting distributed tracing across installed components. Health endpoints means providing installation verification endpoints.

11 Examples

Concrete examples illustrate install style concepts.

11.1 Traditional Unix Application

A typical Unix application following FHS conventions distributes files across the system hierarchy. Executables reside in /usr/bin for user commands and /usr/sbin for system administration commands. Libraries are installed to /usr/lib or /usr/lib64 for shared libraries. Configuration files go in /etc/application for system-wide configuration. Variable data uses /var/lib/application for persistent data and /var/log/application for log files. Documentation is placed in /usr/share/doc/application and man pages in /usr/share/man.

Installation decisions include whether to use /usr (package-managed) or /usr/local (locally installed), how to handle configuration file upgrades (preserve local changes), and what permissions apply to each category.

11.2 Windows Desktop Application

A Windows desktop application typically uses Program Files directories for executables and libraries (in a subdirectory under C:\Program Files). ProgramData stores shared application data (C:\ProgramData\Application). Per-user data goes in AppData directories (Local for machine-specific, Roaming for synchronized). Registry entries store configuration (HKLM for system, HKCU for user). Start menu and desktop shortcuts provide user access.

Installation decisions include 32-bit versus 64-bit Program Files, per-machine versus per-user installation, and User Account Control compatibility.

11.3 Containerized Microservice

A containerized application organizes files within the container image. Application code typically resides in /app or similar directory. Dependencies are installed via package manager or copied directly. Configuration is externalized, mounted at runtime or provided via environment variables. Data volumes mount external storage for persistent data. Entry point scripts initialize and launch the application.

Installation decisions include base image selection, layer optimization for caching, user permissions within container, and read-only root filesystem support.

11.4 macOS Application Bundle

macOS applications use the bundle structure, a directory with .app extension treated as a single unit by Finder. Contents directory contains all bundle contents. MacOS subdirectory contains the main executable. Resources holds assets, localizations, and supporting files. Frameworks contains embedded frameworks. Info.plist declares bundle metadata.

Supporting files outside the bundle include /Library/Application Support for system-wide data, user Library for per-user data, and Preferences for configuration plists.

Installation decisions include whether to embed or reference system frameworks, code signing requirements, and App Store versus direct distribution implications.

11.5 Serverless Function

Serverless functions have minimal installation footprint. Function code is a single file or directory of source code. Dependencies are bundled with the function or provided as layers. Configuration is provided via environment variables or external services. No persistent file system exists; temporary storage only during execution.

Installation decisions include dependency bundling versus layer references, cold start optimization through minimal package size, and external configuration service integration.

12 Best Practices

Experience suggests several best practices for installation architecture.

12.1 Follow Platform Conventions

Adhering to platform conventions provides predictability for administrators, compatibility with platform tools, easier integration with other software, and reduced documentation burden.

Even when conventions seem suboptimal, following them typically outweighs the benefits of custom organization. Deviations should be rare and well-justified.

12.2 Separate Code from Configuration from Data

Clear separation of concerns in file organization provides independent update cycles (code updates without losing configuration), environment-specific configuration (same code, different configuration per environment), backup efficiency (back up data and configuration, not code), and security boundaries (different permissions for different categories).

The twelve-factor app methodology emphasizes this separation, particularly the externalization of configuration.

12.3 Design for Automation

Installation should be fully automatable. Use declarative specifications over imperative scripts where possible. Ensure idempotency so that repeated installation yields the same result. Support unattended installation without interactive prompts. Provide verification capability to confirm installation correctness.

Manual installation procedures do not scale and introduce inconsistency.

12.4 Plan for Updates from the Start

Update scenarios should influence initial installation design. Preserve configuration during updates so that user customizations survive. Support rollback to enable recovery from problematic updates. Consider migration for data format changes between versions. Minimize downtime through appropriate update strategies.

Retrofit consideration for updates is significantly harder than initial design.

12.5 Document the Rationale

Beyond documenting what is installed where, document why. Explain why particular locations were chosen. Note the implications of the organization for operations. Document constraints that influenced decisions. Provide guidance for extending the installation.

Rationale documentation enables informed future modifications.

12.6 Minimize Installation Footprint

Smaller installations are easier to manage. Include only necessary files in the distribution. Use dependencies rather than bundling when appropriate. Clean up build artifacts not needed at runtime. Consider lazy installation for optional components.

Large installations slow deployment, consume resources, and expand attack surface.

12.7 Support Multiple Installation Scenarios

Different deployment contexts may have different needs. System-wide versus user installation should both be supported where appropriate. Provide options for custom installation locations when feasible. Consider containerized versus traditional deployment. Support both interactive and automated installation.

Flexibility in installation enables broader adoption and adaptation.

13 Common Challenges

Installation architecture involves navigating several common challenges.

13.1 Dependency Management

Software dependencies complicate installation. Version conflicts arise when different components require incompatible versions of dependencies. Diamond dependencies occur when multiple paths lead to the same dependency with different version requirements. Transitive dependencies mean dependencies have their own dependencies, expanding the graph. Platform availability concerns whether required dependencies exist on target platforms.

Strategies include bundling dependencies to avoid conflicts at the cost of size, using virtual environments to isolate dependency sets, employing container isolation to provide complete dependency control, and leveraging language-specific package managers that handle dependency resolution.

13.2 Configuration Drift

Installed configurations tend to diverge from intended state over time. Manual changes accumulate as administrators make undocumented modifications. Different environments have different configurations that should be different but may drift unintentionally. Updates may have unintended effects on configuration.

Strategies include configuration management tools that detect and remediate drift, immutable infrastructure that replaces rather than modifies systems, centralized configuration services that externalize configuration, and audit logging to track configuration changes.

13.3 Permission Complexity

File permissions create installation challenges. Installation privileges may require elevated privileges unavailable in all contexts. Runtime permissions must balance security with functionality. Cross-platform differences mean permission models differ across platforms. Container permissions add user namespace mapping complexity.

Strategies include principle of least privilege for all installed files, clear documentation of permission requirements, testing installation with minimal privileges, and using containerization to standardize permission models.

13.4 Path Sensitivity

File paths create portability and maintenance challenges. Hardcoded paths break when installations move. Relative versus absolute paths each have appropriate uses. Path length limits affect deeply nested structures. Case sensitivity varies across file systems.

Strategies include using configuration for all paths rather than hardcoding, preferring relative paths where possible, testing on case-sensitive and case-insensitive file systems, and monitoring path lengths in CI/CD pipelines.

13.5 Upgrade Complexity

Upgrading installed software introduces complexity. Schema migration means data formats may change between versions. Configuration compatibility means new versions may not accept old configuration. Partial upgrade occurs when some components upgrade while others remain at old versions. Rollback means reverting a failed upgrade can be difficult.

Strategies include versioned data schemas with migration support, configuration versioning and migration, atomic upgrade mechanisms, and maintaining previous version for rollback.

13.6 Multi-Platform Support

Supporting multiple platforms multiplies complexity. Convention differences mean each platform has different expectations. Path separator differences exist (forward versus backslash). Tooling differences mean installation tools vary by platform. Testing matrix creates multiplicative testing requirements.

Strategies include abstraction layers to hide platform differences, conditional installation logic for platform-specific elements, containerization for platform-independent deployment, and CI/CD pipelines testing all target platforms.

14 Conclusion

The install style provides essential vocabulary and concepts for documenting how software artifacts map to file system structures. By explicitly capturing software elements, environmental elements, and the relationships between them, install views enable critical analysis of security, maintainability, portability, and operational efficiency.

Effective installation architecture requires balancing multiple concerns: following platform conventions while meeting application-specific needs, separating concerns while maintaining cohesion, enabling automation while supporting customization, and optimizing for common cases while handling edge cases gracefully.

As deployment models evolve—from traditional installations through virtualization to containerization and serverless computing—the install style adapts while maintaining its fundamental purpose: documenting the physical organization of software within its execution environment.

Investment in thoughtful installation architecture pays dividends throughout the software lifecycle, reducing operational burden, improving security posture, enabling reliable updates, and supporting the continuous delivery practices that modern software demands.

References

- Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (2010). *Documenting Software Architectures: Views and Beyond* (2nd ed.).

Addison-Wesley Professional.

- The Linux Foundation. (2015). *Filesystem Hierarchy Standard* (Version 3.0). The Linux Foundation.
- Wiggins, A. (2017). *The Twelve-Factor App.* Retrieved from https://12factor.net/
- Morris, K. (2020). *Infrastructure as Code* (2nd ed.). O'Reilly Media.
- Turnbull, J. (2014). *The Docker Book: Containerization is the New Virtualization.* James Turnbull.
- Microsoft. (2021). *Application Installation on Windows.* Microsoft Documentation.
- Apple. (2021). *Bundle Programming Guide.* Apple Developer Documentation.