MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Distributing standalone Python applications

Bachelor's Thesis

RICHARD ČECH

Brno, Spring 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Distributing standalone Python applications

Bachelor's Thesis

RICHARD ČECH

Advisor: doc. RNDr. Zdeněk Matěj, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools: Grammarly for grammar check along with Github Copilot and ChatGPT for faster code writing and an assistance with debugging and code documentation.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Richard Čech

Advisor: doc. RNDr. Zdeněk Matěj, Ph.D.

Acknowledgements

I want to thank my consultant, RNDr. Jaroslav Čechák, Ph.D., for his regular consultations and expert feedback.

Abstract

This thesis addresses the challenge of distributing Python applications on Windows platforms lacking a pre-installed interpreter. It commences with an analysis of existing distribution methods—including manual setups, packaging tools, and scripted installers—critically evaluating them against key operational criteria such as privilege requirements, update efficiency, package size, and developer overhead. Building on this analysis, a novel PowerShell-based solution is proposed, leveraging an embedded Python interpreter and Inno Setup to generate a user-friendly installer. This custom solution enables user-space installation of a specific Python version alongside the target application, supports efficient incremental updates with minimal user intervention, and provides a native Windows installation experience. The approach is demonstrated using a sample Python application, and its effectiveness is systematically compared with existing methods to highlight its practical benefits in enterprise deployment scenarios and to identify remaining limitations and areas for future work.

Keywords

Python application deployment, Self-contained applications, Software deployment, Windows installer generation, Embedded Python interpreter, PowerShell automation, Application packaging, Dependency management, Enterprise software deployment, User-space deployment, Incremental software updates, Environment isolation

Contents

Introduction **1**

1 Problem Definition and Initial Approaches **2**
1.1 Problem Statement 2
1.2 Naive Approach: Manual Installation Instructions . . . 2
1.3 Partial Improvements: Automated installation 3
1.4 Significance of the Study 4
1.5 Methodological Overview 5

2 Existing Tools and Frameworks **6**
2.1 PyInstaller 6
2.1.1 Internal Architecture and Process 6
2.1.2 Onefile vs. Onedir Distribution 7
2.1.3 Concluding Remarks 8
2.2 PyUpdater 9
2.2.1 Overview and Architecture 9
2.2.2 Internal Mechanism and Workflow 9
2.2.3 End of Official Support and Community Response 9
2.2.4 Concluding Remarks 11
2.3 Chocolatey 11
2.3.1 Overview and Workflow 11
2.3.2 Internal Mechanism 11
2.3.3 Concluding Remarks 13
2.4 cx_Freeze 14
2.4.1 Core Functionality 14
2.4.2 Technical Architecture and Packaging Process . 14
2.4.3 Concluding Remarks 15
2.5 Nuitka 16
2.5.1 Overview and Functionality 16
2.5.2 Technical Architecture 16
2.5.3 Concluding Remarks 18
2.6 Briefcase 18
2.6.1 Overview and Framework Integration 18
2.6.2 Architecture and Build Process 18
2.6.3 Concluding Remarks 20

2.7 Docker 21
2.7.1 Containerization Principles and Workflow . . . 21
2.7.2 Licensing: Docker Desktop vs. Docker Engine . 22
2.7.3 Docker Engine for Windows 23
2.7.4 Concluding Remarks 23
2.8 Summary 23

3 Custom Solution - PowerShell-Based Installer with Embedded Python 25
3.1 Architecture Design Rationale 25
3.2 Underlying Principles and Workflow 27
3.2.1 Packaging Script (Python) 27
3.2.2 Host Setup Script (PowerShell) 28
3.2.3 Inno Setup Installer and Automated Build Scripts 31
3.3 Summary 32
3.4 Key Advantages and Implementation Remarks 34

4 Comparison of Existing Solutions 36
4.1 Evaluation Context and Parameters 36
4.2 Detailed Parameter Analysis 37
4.3 Competitive Positioning 40
4.4 Comparative Analysis 41

5 Future Work 43
Digital Signing and Security 43
Rollback and Recovery Mechanisms 43
Customizable Installer Appearance 43
Graphical User Interface for Developers 44

6 Conclusion 45

Bibliography 46

A Appendix 49
A.1 Source Code Repository 49

List of Tables

4.1 Installation Requirements 41
4.2 End-User Experience 42
4.3 Maintenance and Operations 42

List of Figures

3.1 Component diagram of the build system (Developer's Machine). 26
3.2 Component diagram of the runtime setup (End-User's Machine). 27
3.3 Sequence diagram illustrating the workflow of `setup.ps1`. 30
3.4 Activity diagram of End-to-End Build and Installation Workflow 33

Introduction

Python is currently one of the most widely used programming languages, with applications in data science, machine learning, and automation. Its popularity is driven by its simple syntax, extensive standard library, and availability of numerous third-party packages. Despite these advantages, distributing Python applications to end users remains a nontrivial task, particularly on Windows systems where the Python interpreter is not installed by default.

In typical deployment scenarios, users must manually install the interpreter, manage package dependencies, and address potential compatibility issues [1]. This process may present significant obstacles, especially in environments where users lack technical expertise or do not have administrative rights. As a result, Python-based applications are often less accessible to their intended audience than equivalent applications built in compiled languages [2].

This thesis aims to design and demonstrate a solution for distributing Python applications on systems without a pre-installed interpreter. The proposed approach produces a single executable file that installs a specific version of the Python interpreter and the application itself. In addition, it incorporates automatic update support, allowing the application to remain current with minimal user intervention.

Existing distribution methods are reviewed and evaluated to support the proposed solution, highlighting their limitations in usability, scalability, maintainability, and workspace environment consistency. A demonstration is provided using a sample Python application that shows the practical benefits of the approach while also acknowledging its remaining limitations. The aim is to reduce the technical barrier associated with deploying Python software and to improve the overall accessibility and maintainability of Python-based applications in user-facing contexts.

1 Problem Definition and Initial Approaches

Distributing Python applications to systems without a pre-installed interpreter presents several practical and technical challenges. While Python is well-regarded for its developer productivity and extensive library support, its interpreted nature complicates deployment, particularly for end users without technical experience or in environments with limited configuration options.

This chapter examines common strategies to address these challenges, beginning with basic manual approaches and progressing toward partially automated methods. The aim is to highlight the shortcomings of existing solutions and establish the need for a more streamlined distribution mechanism, which is presented in Chapter 3.

1.1 Problem Statement

Unlike compiled languages that produce standalone binaries, Python applications require an interpreter. On systems where Python is not pre-installed, deployment becomes nontrivial. Users must typically install the interpreter manually, resolve dependencies through package managers, and ensure version compatibility between components [3]. These steps introduce a high risk of inconsistency between environments, increased support overhead, and a poor user experience, particularly for non-technical users. This limits the accessibility of Python-based software in practical, user-facing scenarios.

1.2 Naive Approach: Manual Installation Instructions

A widely adopted approach to Python application distribution involves including detailed setup instructions in a README file, typically provided with the application [1]. This document outlines the steps to manually install the Python interpreter, resolve package dependencies, and configure the execution environment. Provided the instructions are accurate and comprehensive, this method may be effective for users with sufficient technical competence.

However, this approach has several significant drawbacks:

- **Extensive user intervention:** The installation process requires users to execute multiple steps manually, increasing the demands of user expertise and likelihood of errors, further reducing overall usability.
- **Dependency management issues:** Manual installation of packages can lead to inconsistent dependency versions across systems, resulting in unpredictable application behavior [3].
- **Interpreter version conflicts:** Users may be required to install specific versions of the Python interpreter, potentially leading to conflicts between multiple installations [4].
- **Low maintainability:** Updates to the application or its dependencies typically require users to repeat the installation process, which can be time-consuming and error-prone.
- **Increased support demands:** Developers may be required to provide ongoing assistance due to variations in user environments and levels of expertise [2].

Although this method is conceptually simple, it does not provide a reliable or scalable solution for distributing Python applications to a broad user base.

1.3 Partial Improvements: Automated installation

To address some of the limitations associated with manual installation, developers may utilize automated installation mechanisms. These include scripted installers or traditional installation packages that programmatically download and install the required Python interpreter and application dependencies using tools such as `pip` [5].

Automated installation reduces the need for user intervention and improves the overall consistency of the deployment process. In particular, it simplifies initial setup and subsequent updates, improving maintainability. Nevertheless, this approach remains subject to several constraints. It often requires users to possess installation privileges, assumes the availability of system dependencies, and may not eliminate issues related to interpreter versioning and dependency conflicts [6].

Some developers employ custom installation scripts to streamline the setup process; however, these scripts are often fragile, raise security concerns, and lack cross-platform reliability [7]. While these approaches represent a practical improvement over fully manual procedures, they do not offer a fully self-contained or universally robust solution for Python application distribution.

1.4 Significance of the Study

The distribution of Python applications presents significant practical challenges, particularly in corporate environments where end users have limited technical expertise or restricted system privileges. This thesis addresses real-world deployment issues encountered during a collaboration with industry partners, where existing distribution methods proved inadequate for operational needs.

In production environments, the limitations of current approaches become particularly evident: PyInstaller creates monolithic executables that complicate incremental updates; Chocolatey requires administrative rights, often unavailable to standard users; Docker introduces operational overhead inappropriate for desktop applications. Organizations face concrete business challenges when Python-based tools cannot be efficiently deployed to end users, including increased IT support costs, delayed software rollouts, and reduced productivity.

A particularly problematic aspect observed in corporate settings is the need to reinstall entire Python environments with each application update. This creates substantial bandwidth consumption, extends deployment timelines, and increases the risk of installation failures, all of which translate to tangible business costs and operational disruptions.

This thesis addresses these industry-validated challenges by proposing a distribution method within standard user permission boundaries while eliminating dependency on pre-installed interpreters. The solution delivers incremental update capabilities, significantly reducing bandwidth consumption and deployment time compared to traditional approaches. Integrating with familiar Windows application management conventions minimizes training requirements while maintaining consistency across varying corporate workstation configurations. The methodology's practical significance stems from its

development in response to business requirements, tested against real-world constraints and use cases.

Solving these concrete deployment challenges enables organizations to leverage Python-based applications more effectively across various operational contexts—from internal business tools to customer-facing utilities—without compromising on security, efficiency, or user experience [8]. This industry-informed approach offers substantial value for Python applications deployed in corporate, educational, or research environments where IT resources are constrained and technical support capacity is limited. The solution aims to bridge the gap between Python's development advantages and the practical realities of enterprise software distribution.

1.5 Methodological Overview

To achieve the stated objective, this thesis adopts the following methodological structure:

- **Analysis of existing approaches:** A review and evaluation of current distribution methods, focusing on their implementation, usability, and limitations.
- **Design of a custom solution:** A proposal for a self-contained distribution mechanism capable of bundling the application and a specific Python interpreter version into a single executable package.
- **Implementation:** Development of the proposed solution and demonstration using a representative Python application.
- **Evaluation:** A qualitative comparison between the proposed and existing solutions, considering criteria such as ease of installation, update capabilities, and user accessibility.

2 Existing Tools and Frameworks

The complex challenge of distributing Python applications in a fully self-contained manner has persisted for many years, leading to various tools and frameworks designed to simplify the deployment. This chapter overviews the most widely used solutions, examining their core features, strengths, and limitations. This discussion aims to clarify how each tool addresses the complexities of packaging, updating, and distributing Python software.

2.1 PyInstaller

PyInstaller is one of the most popular tools for transforming Python applications into standalone executables, supporting Windows, Linux, and macOS platforms [9]. Unlike traditional installation methods, which require users to install a Python interpreter and manage dependencies, PyInstaller embeds the Python interpreter and all necessary libraries directly into the distributable package. This approach dramatically reduces the risk of missing dependencies and simplifies the end-user experience, as applications can be run without any prior Python setup.

2.1.1 Internal Architecture and Process

PyInstaller analyzes the specified Python script, recursively identifying all imported direct and indirect modules and assembling a comprehensive list of dependencies. The tool then bundles these modules, third-party packages, and the Python interpreter into a single executable artifact. The main components of this process include:

- **Bootloader:** A minimal, platform-specific executable written in the C programming language, responsible for initializing the embedded Python interpreter and launching the packaged application code.
- **Dependency Analysis:** A sophisticated discovery phase that constructs a dependency graph by tracking imports at both the compile-time and runtime. PyInstaller performs static analysis

on the code and executes parts in a controlled environment to detect dynamically loaded modules. This comprehensive approach ensures that even modules imported using techniques like `importlib` or `__import__()` are captured.

- **`.spec` File:** A configuration file generated by PyInstaller, which records build instructions such as files to exclude, additional data to include, and custom hooks. Developers can modify this file to fine-tune the packaging process.

2.1.2 Onefile vs. Onedir Distribution

PyInstaller offers two main packaging formats. The *onefile* mode compresses all components into a single executable, making distribution and installation straightforward for users. However, this convenience comes at the cost of slower startup times, as the executable must extract its contents into a temporary directory before launching. This delay can be noticeable for applications with significant dependencies and external resources.

In contrast, the *onedir* mode produces a directory that contains the executable, interpreter, libraries, and user code. Although this format is less convenient to distribute as a single file, it allows for faster application startup and easier access to bundled resources, which can be beneficial during development or for applications that require frequent updates.

Advantages

- **No Pre-installed Interpreter Required:** The application runs independently of any system Python installation, reducing compatibility issues.

- **Ease of Use:** The straightforward command-line interface requires minimal configuration for basic applications, and the tool handles most common scenarios without extensive customization. This makes it accessible even to developers with limited packaging experience.

- **Flexible Distribution Options:** Developers can choose between a single file executable for ease of sharing or a directory-based structure for performance and transparency.

Disadvantages

- **Large Output Size:** Bundling the interpreter and all dependencies results in sizable packages, which may be problematic for bandwidth-limited environments. Even simple applications often produce executables that exceed 10-15 MB due to the included Python runtime.

- **Lack of Incremental Updates:** PyInstaller does not natively support patching or differential updates, so distributing new versions requires users to download the entire package again, regardless of how minor the changes may be.

- **Slow Startup Time:** Particularly in one-file mode, applications can experience noticeable delays during launch, as the package extracts files to a temporary location before execution. This overhead increases proportionally with the size of the included libraries.

- **Debugging Challenges:** Troubleshooting issues in packaged applications is more complex than with standard Python scripts, as error messages may be cryptic and the bundled environment differs from development.

2.1.3 Concluding Remarks

Overall, PyInstaller provides a practical and widely adopted solution to create standalone executables from Python code [9]. Its cross-platform support and straightforward workflow make it accessible to developers and end users. However, package size and update efficiency tradeoffs should be considered, especially for projects anticipating frequent releases or requiring minimal download sizes.

2.2 PyUpdater

2.2.1 Overview and Architecture

PyUpdater was created to streamline the process of delivering updates for Python applications, particularly those packaged with PyInstaller [10]. In this model, developers host new releases on a remote server, and the application checks for updates either at startup, predefined intervals, or through manual user requests. If an update is available, PyUpdater downloads and applies it, optionally using delta patches to minimize bandwidth usage. All update files are cryptographically signed using asymmetric encryption to ensure integrity and protect against tampering.

2.2.2 Internal Mechanism and Workflow

PyUpdater integrates a client library into the packaged application. When triggered, this client connects to a remote endpoint hosting a manifest file—typically in JSON format—that enumerates available versions, file hashes, and cryptographic signatures. The client compares the local version against those available remotely and initiates the update upon detecting a newer release.

For updates, the client can download a new package or retrieve a binary delta patch containing only the differences between versions. This patching mechanism, particularly valuable for large applications, can reduce download sizes by up to 90% compared to full package updates [10]. After retrieving the update files, the client verifies their integrity using the embedded public key, extracts or applies patches as needed, and prompts for application restart or handles it automatically.

2.2.3 End of Official Support and Community Response

Official development and support for PyUpdater ceased in December 2022, as acknowledged by the project maintainers in their final repository update [11]. This discontinuation created compatibility concerns as Python and PyInstaller continued to evolve with new releases.

In response, the community developed *PyUpdater2*, a fork that aims to maintain compatibility with recent Python versions while resolving outstanding bugs from the original implementation [12].

Although this fork preserves the core functionality of PyUpdater, it operates with volunteer maintenance and lacks formal enterprise support guarantees. Organizations considering PyUpdater2 for production deployment should evaluate the implications of relying on community-supported infrastructure for their update pipelines.

Advantages

- **Automated Update Checks:** Applications can automatically detect and apply updates without user intervention, improving both user experience and security posture by ensuring that vulnerability fixes are promptly deployed. The update frequency can be configured based on application needs, from checking at every launch to periodic background verification.

- **Delta Updates:** By distributing only the binary differences between versions, PyUpdater significantly reduces bandwidth requirements—often by 70-90% compared to full package updates. This efficiency is particularly valuable for applications with large resource files or frequent minor updates.

- **Cryptographic Verification:** Updates are signed with Ed25519 public-key cryptography, providing strong protection against tampering and ensuring that only authorized updates from the original developer are installed.

Disadvantages

- **Reliance on Community Forks:** The cessation of official support increases the risk of incompatibility with future Python releases. While PyUpdater2 is actively maintained, its community-driven nature may introduce uncertainty for mission-critical applications.

- **Larger Initial Package Size:** Bundling the embedded Python interpreter and all dependencies increases the size of the installer compared to solutions that rely on a system-wide Python installation.

- **Additional Application Complexity for Updates:** To support features like automated update checks or incremental (delta) updates, developers must explicitly implement and maintain this logic within the application codebase. This increases development effort and can introduce maintenance challenges.

- **Hosting Infrastructure Required:** Developers must manage and secure a remote server for update distribution, which can be a barrier for smaller teams.

2.2.4 Concluding Remarks

PyUpdater introduced an efficient and secure update mechanism for Python applications, but its archival status presents challenges for long-term use. The emergence of PyUpdater2 offers a viable alternative for teams needing automated updates, though reliance on community support should be carefully considered.

2.3 Chocolatey

2.3.1 Overview and Workflow

Chocolatey is a package manager tailored for Windows, enabling automated installation, upgrading, and removal of software through scriptable packages [13]. Unlike tools that bundle Python within the application, Chocolatey operates at the system level, managing both applications and their prerequisites. For Python-based software, this typically means Chocolatey first ensures an appropriate Python interpreter is available on the system, then installs the application and its dependencies. This approach aligns with traditional package management philosophies found in Linux distributions but adapted for the Windows ecosystem.

2.3.2 Internal Mechanism

Chocolatey packages are ZIP archives with a .nupkg extension, containing a manifest (.nuspec) and PowerShell scripts (chocolateyInstall.ps1, and optionally chocolateyUninstall.ps1). When a user runs

`choco install <package>`, Chocolatey resolves dependencies, downloads the necessary packages, and executes the installation scripts. These scripts can perform various actions, such as extracting installers, configuring the system, or validating checksums.

For Python applications, these scripts typically handle installing the required Python version if not already present, configuring environment variables, and using pip to install the application's dependencies. Unlike bundling approaches, this system-wide installation method means Python and its libraries become available for other applications, potentially saving disk space but introducing possible version conflicts between different software packages.

Upgrades and removals are handled by executing the corresponding scripts, ensuring that installations remain consistent and repeatable. The package-based approach allows for centralized management across an organization, making it particularly suitable for enterprise environments where standardization is valued.

Advantages

- **Windows Integration:** Chocolatey fits seamlessly into the Windows environment, administration workflows, supporting centralized management and deployment across multiple machines.
- **Automated Dependency Management:** Packages can specify dependencies (including Python versions) that are automatically resolved during installation, reducing complex setup steps for end users.
- **Scripted Deployment:** The PowerShell-based automation supports unattended installations and configuration management tools, making large-scale deployments more efficient.

Disadvantages

- **No Self-Contained Interpreter:** Moreover, unlike self-contained approaches, Chocolatey installations affect the global system environment, potentially introducing conflicts between applications that require different Python versions or package dependencies.

- **Potential for Environment Conflicts:** Relying on shared dependencies increases the risk of incompatibility, primarily when multiple Python versions exist.

- **Administrative Privileges:** Many operations require elevated permissions, limiting usability in restricted environments where users lack administrative access to their workstations.

- **Significant Developer Overhead:** Creating and maintaining Chocolatey packages demands substantial effort from developers. Each package requires crafting precise PowerShell scripts, understanding NuGet conventions, and managing the publication process—skills not typically part of Python developers' core expertise.

- **Additional Scripting for User Experience:** Users may face complexity during installation and updates without extensive custom scripting. Developers must implement additional automation to manage Python virtual environments, handle upgrade paths, and provide a streamlined experience comparable to native applications.

2.3.3 Concluding Remarks

Chocolatey provides a package management solution for Windows environments, offering benefits for IT-managed deployments through its integration with administrative workflows. However, its fundamental approach—relying on system-wide Python installations rather than bundled runtimes—introduces significant limitations for Python application distribution. The requirement for administrative privileges, potential for dependency conflicts, and substantial developer overhead make it poorly suited for delivering applications to diverse user environments. While valuable in tightly controlled organizational settings, Chocolatey represents a challenging option for Python developers seeking to distribute applications to users with varying technical capabilities or restricted permissions.

2.4 cx_Freeze

2.4.1 Core Functionality

cx_Freeze transforms Python applications into standalone directory-based packages that can run on systems without a Python installation [14]. This cross-platform tool follows Python's standard packaging conventions, allowing developers to define their build configuration in familiar formats like `setup.py` or `pyproject.toml`. The straightforward build process—typically a single command such as `python setup.py build`—produces a distribution folder containing the application code, dependencies, and a platform-specific Python runtime.

2.4.2 Technical Architecture and Packaging Process

When building an application, cx_Freeze performs several key operations [14]. First, it analyzes the application's entry point to identify imported modules and dependencies. It then compiles Python source files to bytecode for performance optimization and copies necessary runtime components, including the appropriate Python DLL for the target platform. Finally, it generates a native executable as the application launcher, initializing the embedded Python interpreter with the correct paths and environment configuration. This architecture creates a self-contained package that functions independently of the host system's Python installation.

Advantages

- **Familiar Development Workflow:** cx_Freeze integrates with Python's standard packaging tools and conventions, minimizing the learning curve for Python developers.

- **Debugging Transparency:** The directory-based structure allows developers to easily inspect the packaged files, modify resources, and troubleshoot runtime issues.

Disadvantages

- **Multi-File Distribution:** Unlike PyInstaller's one-file option, cx_Freeze only produces directory-based packages, similar to PyInstaller's onedir mode. This necessitates additional steps to create a seamless installation experience for end users.

- **No Built-In Update Mechanism:** The tool provides no native facilities for checking, downloading, or applying updates, requiring developers to implement these critical features separately or rely on third-party solutions [14].

- **Limited Platform Integration:** While functional across operating systems, cx_Freeze lacks deeper integration with platform-specific features like Windows registry entries or desktop shortcuts, requiring manual implementation.

- **Substantial Package Size:** Including a complete Python runtime significantly increases distribution size, often 20-50MB before application code, which can be problematic for bandwidth-limited scenarios.

2.4.3 Concluding Remarks

cx_Freeze effectively solves the basic distribution problem by creating self-contained Python applications, functioning much like PyInstaller's directory mode but with a more Python-centric configuration approach. Its primary strengths lie in its familiar Python-based build process and transparent output structure, making it accessible to developers who are already comfortable with Python packaging tools. However, its limitations—particularly the lack of single-file packaging, update capabilities, and installer creation—position it as better suited for internal tools and development environments rather than polished consumer applications. Organizations seeking to distribute applications to end users will typically need to combine cx_Freeze with additional packaging tools to create a complete deployment solution.

2.5 Nuitka

2.5.1 Overview and Functionality

Nuitka differentiates itself from other Python packaging tools by functioning as an actual compiler rather than a bundler [15]. It translates Python code into optimized C++ and compiles it into native machine code using platform-specific compilers. This approach aims to improve execution performance while preserving compatibility with standard Python libraries and extensions. The compilation process is initiated through command-line directives, with numerous options for controlling optimization levels and output formats.

2.5.2 Technical Architecture

Nuitka operates through a multi-phase compilation process:

1. **Parsing:** Python source files are analyzed and converted into an abstract syntax tree (AST), similar to how the CPython interpreter processes code initially.
2. **Code Transformation:** The AST is transformed into optimized C++ code that mimics Python's runtime behavior while eliminating interpreter overhead.
3. **Native Compilation:** Generated C++ code is compiled into binary machine code using a platform-specific C++ compiler (MSVC on Windows, GCC/Clang on Unix-based systems).
4. **Linking:** The resulting executable is linked against a minimal runtime library that implements essential CPython features and interfaces with native extension modules.

This compilation strategy preserves Python's dynamic features while removing the performance penalties associated with bytecode interpretation. Unlike traditional Python execution, which interprets bytecode at runtime, Nuitka's approach produces native executables that can achieve significantly faster execution speeds, particularly for computation-intensive operations and frequently executed code paths [16].

Advantages

- **Execution Performance:** By compiling to native code, Nuitka can deliver substantial speed improvements for CPU-bound applications, potentially addressing performance bottlenecks that otherwise require rewriting portions in C/C++.

- **Intellectual Property Protection:** The compilation process obscures source code more effectively than bytecode compilation, providing a degree of protection for proprietary algorithms and business logic.

- **Reduced Startup Time:** Compiled applications typically launch faster than interpreted ones, improving the user experience, particularly for frequently used utilities.

Disadvantages

- **Complex Toolchain Requirements:** Unlike tools like PyInstaller or cx_Freeze, Nuitka requires a functioning C++ development environment, creating an additional technical barrier for developers and potentially complicating continuous integration workflows.

- **No Embedded Update Mechanism:** Similar to other compilation tools, Nuitka provides no built-in functionality for checking, retrieving, or applying updates, requiring developers to implement these critical features separately.

- **Limited Integration with Standard Python Packaging:** Nuitka's compilation approach exists somewhat outside the standard Python packaging ecosystem, reducing compatibility with conventional tools like pip or setuptools.

- **Substantial Developer Overhead:** Debugging compiled applications presents unique challenges, as error messages may differ from those in interpreted Python, and standard debugging tools may be less effective.

2.5.3 Concluding Remarks

Nuitka occupies a specialized niche in the Python distribution landscape, appealing primarily to developers seeking performance optimization or intellectual property protection rather than simplified deployment. While it addresses certain performance limitations inherent to interpreted Python, it introduces complexity that may be disproportionate to the benefits for many applications.

Its lack of integrated update capabilities and requirement for C++ development tools make it less suitable for general-purpose application distribution than bundling solutions like PyInstaller. Nuitka is best considered for computationally intensive applications where performance justifies the additional development and maintenance overhead.

2.6 Briefcase

2.6.1 Overview and Framework Integration

Briefcase is part of the BeeWare project, a suite of tools and libraries to facilitate native, cross-platform Python application development [17]. Unlike packaging tools focusing solely on bundling, Briefcase produces platform-native installers—MSI or MSIX packages on Windows, DMG files on macOS, and appropriate formats for other platforms. Its primary goal is to provide end users with a familiar installation experience comparable to commercial software. The build process is managed through the briefcase command-line interface and configured via standard Python packaging metadata in `pyproject.toml` [18].

2.6.2 Architecture and Build Process

Briefcase's workflow is structured around distinct phases:

1. **Project Initialization:** The `briefcase new` command scaffolds a project structure that follows Python packaging standards while adding platform-specific configuration.

2. **Application Building:** When executing `briefcase create`, Briefcase generates platform-specific project templates. This involves

creating WiX Toolset XML definitions alongside a copy of the application code on Windows.

3. **Environment Preparation:** The `briefcase build` command constructs a dedicated virtual environment containing the Python interpreter and all dependencies specified in the project configuration.

4. **Installer Generation:** Finally, `briefcase package` invokes platform tools (such as the WiX compiler on Windows) to produce installers that bundle the application, its dependencies, and a complete Python runtime.

This structured approach emphasizes seamless integration with each operating system's native installation mechanisms, providing an experience that aligns with users' expectations for professional software.

Advantages

- **Native Integration:** By generating platform-standard installers, Briefcase applications appear and behave like native software, following operating system installation, uninstallation, and presentation conventions.

- **User-Friendly Experience:** End users interact with familiar installation wizards rather than custom or technical interfaces, reducing friction and increasing acceptance in corporate environments.

- **BeeWare Ecosystem Compatibility:** Applications built with BeeWare's Toga GUI framework gain additional integration benefits, enabling truly native interface elements across platforms [19].

Disadvantages

- **No Built-In Update Mechanism:** Unlike tools such as PyUpdater, Briefcase provides no facilities for in-place or incremental updates, requiring complete reinstallation for even minor application changes.

- **Substantial Package Size:** Each installer contains a complete Python runtime and all dependencies, resulting in large distribution files (typically 50+ MB) that consume significant bandwidth during deployment and updates.

- **Significant Developer Overhead:** Creating Briefcase packages requires developers to learn the BeeWare ecosystem, understand platform-specific packaging concepts, and maintain detailed configuration in pyproject.toml. This specialized knowledge increases the learning curve compared to simpler Python packaging tools.

- **Non-Python Tool Requirements:** Unlike pure Python packaging solutions, Briefcase requires installation and configuration of platform-specific build tools (WiX Toolset on Windows, xcodebuild on macOS) that many Python developers may not be familiar with, complicating development workflows and continuous integration pipelines.

- **Limited Customization:** While the native installer approach improves user experience, it restricts developer control over the installation process compared to custom Python-based solutions.

2.6.3 Concluding Remarks

Briefcase excels at creating professional, platform-native installers that provide end users with a familiar experience—a significant advantage in business environments where software must appear polished and trustworthy. However, its approach introduces substantial tradeoffs for deployment scenarios requiring frequent updates or bandwidth-conscious distribution. The combination of large installer sizes and lack of incremental update capabilities makes Briefcase less suitable for rapidly evolving applications or deploying to environments with limited connectivity. For Python applications intended for occasional release with emphasis on professional presentation, Briefcase offers significant advantages, but developers should consider alternative approaches when update efficiency is a priority.

2.7 Docker

Docker is a containerization platform that packages software into lightweight, isolated environments called *containers* [20]. These containers encapsulate the application, its dependencies, and the Python interpreter, ensuring consistent behavior across different systems. This approach fundamentally differs from traditional Python packaging tools by focusing on environment isolation rather than executable creation. For Python applications specifically, containerization prevents the "works on my machine" problem by standardizing the runtime environment, regardless of the host system's configuration [21].

2.7.1 Containerization Principles and Workflow

Docker operates around several core concepts:

- **Images:** Immutable snapshots built from a `Dockerfile`, containing the base operating system, Python interpreter, dependencies, and application code.
- **Containers:** Running instances of images, isolated from the host system but sharing the host's kernel. Each container provides a consistent execution environment for Python applications.
- **Registries:** Repositories (such as Docker Hub) where images are stored and from which they can be pulled for deployment, facilitating versioning and distribution of containerized applications.

For Python applications, developers typically create a `Dockerfile` that starts with a base Python image, installs dependencies using pip, and copies the application code into the container [20]. This produces a reproducible environment that can be deployed consistently across development, testing, and production environments.

Advantages of Docker

- **Consistent and Isolated Runtime:** Applications run identically across different machines, eliminating environment-specific bugs

and simplifying deployment across heterogeneous infrastructure.

- **Incremental Updates:** Docker's layered image model allows users to download only the changed parts of an image, saving bandwidth and enabling efficient deployment of application updates [22].
- **Cross-Environment Support:** Particularly effective for headless or server applications, as well as CI/CD pipelines, ensuring consistent testing and execution environments throughout the development lifecycle.

Disadvantages of Docker

- **Requires Docker Installation:** Users must install Docker Desktop, which may not be feasible in restricted corporate environments or for end users without technical expertise.
- **Limited GUI Support:** Running graphical Python applications in containers requires additional configuration for display server access, making it impractical for desktop applications, especially on Windows.
- **Large Windows Base Images:** Windows containers are significantly larger than their Linux counterparts (often 3-4GB vs. 100-200MB), substantially increasing storage requirements and initial download times.
- **Significant User Overhead:** End users must learn container concepts and Docker commands to manage applications, creating a steeper learning curve than traditional installation methods.

2.7.2 Licensing: Docker Desktop vs. Docker Engine

Docker Desktop, which provides a graphical interface and additional features, introduced licensing requirements for organizations with over 250 employees or annual revenue exceeding $10 million [23]. This change has licensing implications for companies distributing Python applications via Docker. In contrast, Docker Engine remains open

source (Apache License 2.0). It can be installed separately, though it lacks the convenience of the Desktop interface and requires more technical expertise to configure correctly.

2.7.3 Docker Engine for Windows

Using Docker Engine alone on Windows involves manual configuration and relies on large Windows base images. For Python applications specifically, this approach introduces several challenges: Windows containers cannot use the more efficient Linux-based Python images, Python's file path handling differs between Windows and Linux environments, and some Python packages with native components may not function correctly in Windows containers [24]. These limitations make Docker on Windows best suited for headless or service-oriented Python applications rather than desktop software.

2.7.4 Concluding Remarks

Docker excels at creating reproducible, isolated environments for Python applications, particularly in server-side, microservice, and cloud deployment scenarios. Its ability to ensure consistent execution across different systems makes it invaluable for complex Python applications with numerous dependencies. However, for distributing Python desktop applications to end users, Docker presents significant barriers: the requirement for Docker installation, complex GUI handling, licensing considerations, and the technical expertise needed to run containerized applications. When selecting Docker as a distribution strategy for Python applications, organizations should carefully evaluate these tradeoffs, considering the technical requirements and the target user base's capabilities.

2.8 Summary

The tools reviewed in this chapter address different aspects of standalone Python distribution, from single-file executables and directory-based deployments to automated updating and containerization:

- **PyInstaller:** Excels at creating standalone executables that bundle the Python interpreter with application code, but produces large packages and lacks efficient update mechanisms [9].
- **PyUpdater and PyUpdater2:** Offer automated update capabilities for Python applications, but require additional infrastructure and do not reduce initial package size [10].
- **Chocolatey:** It is well-suited for enterprise Windows environments with its package management approach, but it depends on system-wide Python installations and requires administrative privileges [13].
- **cx_Freeze:** Provides self-contained distributions in a directory structure, using a familiar Python-centric build process, but lacks single-file packaging and update capabilities [14].
- **Briefcase:** Creates native installers with a professional user experience, but results in large packages and lacks integrated update features [18].
- **Nuitka:** Offers performance benefits through compilation to native code, but introduces build complexity and requires C++ development tools [15].
- **Docker:** Stands out for server and CI/CD deployments with its consistent environments and incremental updates [20], but remains less suitable for desktop applications due to installation requirements and image size.

Collectively, these solutions highlight the tradeoffs between portability, ease of updating, resource efficiency, and user experience. The ongoing evolution of Python packaging tools reflects the diverse needs of developers and organizations, underscoring the importance of selecting the right approach based on specific project requirements and deployment contexts [8].

3 Custom Solution - PowerShell-Based Installer with Embedded Python

A custom solution is required to address the individual shortcomings of both the naive approach and the existing tools. This tailored approach effectively mitigates or, in some cases, eliminates the disadvantages associated with distributing standalone Python applications. The resulting solution offers a more universal and streamlined method for packaging Python programs, allowing them to run independently of a pre-installed Python interpreter on the host system. This makes it a practical and scalable tool suitable for various deployment scenarios.

3.1 Architecture Design Rationale

As illustrated in Figure 3.1 and Figure 3.2, the solution architecture is split between the build system on the developer's machine and the runtime system on the end-user's machine. This separation addresses key limitations identified in existing Python distribution tools.

- **Package Size vs. Independence:** Tools like PyInstaller create fully independent executables at the cost of large file sizes that hamper distribution and updates.
- **Update Mechanism Gaps:** Most packaging solutions lack built-in update capabilities, with PyUpdater being the exception, but requiring significant infrastructure.
- **Administrative Privileges:** Solutions like Chocolatey provide robust integration but require administrative rights, often unavailable in restricted environments.
- **Developer Experience:** Many approaches demand specialized knowledge outside typical Python development skills, increasing the maintenance burden.
- **User Experience:** The end-user installation process often involves technical steps or unfamiliar interfaces that reduce accessibility.

This custom solution's architecture leverages the concept of embedded Python interpreters, which provide independence without full bundling overhead, combined with familiar Windows installation paradigms. By separating the distribution into distinct components (packaging, installation, and update mechanisms), the architecture achieves several key design objectives:

First, it maintains user space operation, eliminating the need for administrative privileges while still providing a professional installation experience. Second, it enables incremental updates by separating the Python interpreter from application code, allowing for smaller patch downloads. Third, it uses standard Windows installer technology to provide users with familiar interfaces while utilizing Python's ecosystem for the developer workflow.

The architecture deliberately avoids cross-platform compatibility in favor of optimizing for Windows, where the interpreter distribution challenge is most acute. This decision allows for deeper integration with Windows-specific mechanisms like registry entries and standard installer patterns, resulting in a more polished end-user experience for the target platform.

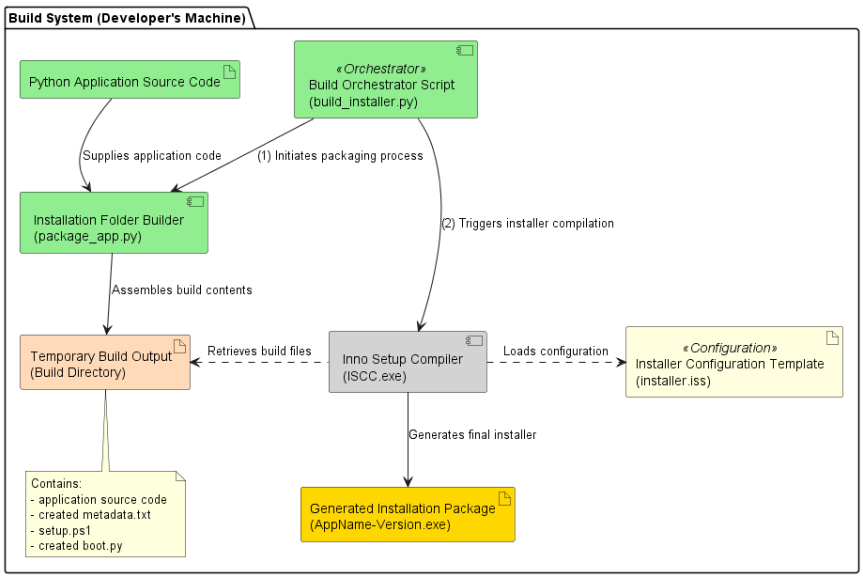


Figure 3.1: Component diagram of the build system (Developer's Machine).

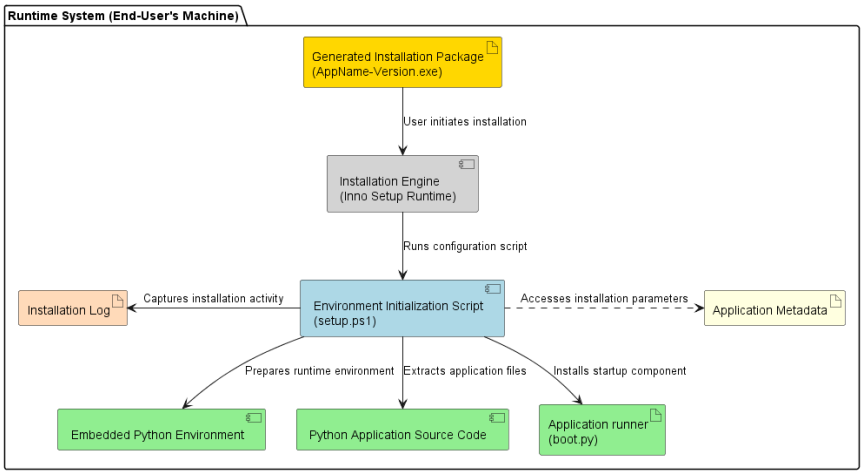


Figure 3.2: Component diagram of the runtime setup (End-User's Machine).

3.2 Underlying Principles and Workflow

The core idea behind the proposed solution is to provide the smallest possible package to the end user while preserving key advantages such as usability and reliability. This is achieved using an embedded Python interpreter that removes the need for a complete Python installation on the user's system. Combined with well-structured scripting and appropriate tools, the solution is lightweight, easy to use, and functionally comparable to conventional packaging methods. The approach is built upon four interlocking components:

3.2.1 Packaging Script (Python)

The `package_app.py` script builds a deployment folder ready to be processed by the Inno Setup Compiler (ISCC). This script establishes a fixed folder structure and ensures all application components are in place before the final installer is generated. Its responsibilities include:

1. **Source Code Collection:** The script copies the complete application source tree from a specified directory into a target folder. It

filters out unnecessary files (e.g., `__pycache__`, `.pyc`, `.git`, and `.vscode`) to maintain a clean package.

2. **Python Version Determination:** The helper function `get_python_version()` reads the attached `requirements.txt` file and uses regular expressions to extract the specified Python version. If the version is below 3.9 or the file is missing, the script warns the user and falls back to a default version (3.10.0). This ensures that the embedded Python interpreter is compatible with the application.

3. **Internal Folder and Metadata Generation:** The script creates an internal directory (named `_internal`) within the build folder. This directory hosts vital installation files, including:

   - **Metadata File:** A `metadata.txt` file is generated containing key properties such as the application name, installation folder name, entry file, version, and determined Python version. This metadata guides the installer's behavior.
   - **Boot Script:** The `create_boot_py()` function produces the `boot.py` file, embedding constants (for the Python environment and logs directories) and configuring the execution path. This script is essential for launching the application on the end-user side with the self-contained Python interpreter.
   - **Custom Path File:** A `custom_pth.txt` file is created (via `generate_custom_pth()`) to correctly set up the Python environment path for further `pip` integration.

4. **Copying Installer Script:** Alongside the application files, the script copies another essential installation script - `setup.ps1` into the internal folder. A helper batch file (`setup.bat`) is also generated to invoke the PowerShell setup during installation.

3.2.2 Host Setup Script (PowerShell)

The `setup.ps1` script is the backbone of the installation and update process. Developed in PowerShell, this script runs on the developer's

machine and performs the following concrete tasks (as illustrated in the sequence diagram in Figure 3.3).

- **Parameter Initialization and Logging:** The script accepts parameters such as `-InstallPath` and `-CurrentInstalledVersion`. It immediately creates a robust logging mechanism with a global log file and early-fallback logging to capture initialization errors.
- **Installation Directory Validation:** It validates the provided installation path, ensuring that it exists and is accessible. If the path is invalid, the script logs an error and exits.
- **Metadata Processing and Version Checks:** The script reads the `metadata.txt` file from its internal folder (`_internal`) to extract essential details (such as `AppName`, `Version`, `AppFolder`, and `PythonVersion`). It compares the version against the currently installed version, determining whether an update or a clean install is necessary. In update scenarios, if the new version is the same or older than the current install, it triggers a clean install wipe.
- **Conditional Python Environment Setup:** Depending on the version check and the state of the existing environment, the script decides whether to perform a full Python setup. This includes removing any outdated Python environment, downloading the correct version of the Python embeddable package, extracting it into the target folder, and replacing the `._pth`.
- **Ensuring `pip` and Dependency Installation:** Once the embedded Python environment is prepared, the script ensures that `pip` is installed. It downloads and executes `get-pip.py` if necessary, upgrades core build tools (like `setuptools` and `wheel`), and finally installs or updates dependencies based on the `requirements.txt` file from the application folder.
- **Error Handling and Finalization:** Throughout the process, detailed logging captures every step and error condition. The script includes fallback mechanisms for logging failures and provides clear messages if a critical step (e.g., download of Python or

dependency installation) fails. The script logs the finish status on successful completion and exits with a success code.

Leveraging PowerShell's native Windows integration and strong automation capabilities, `setup.ps1` orchestrates complex installation tasks—from directory validation to handling embedded Python updates—ensuring a reliable, repeatable, and user-transparent setup process.

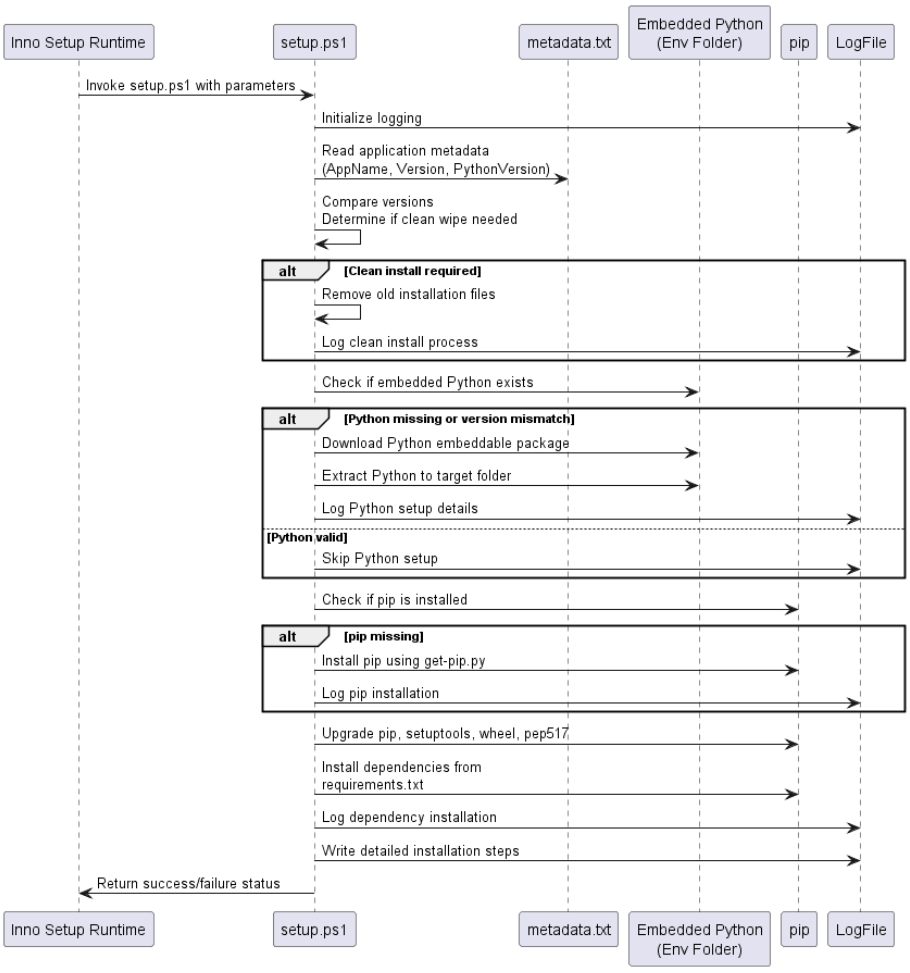


Figure 3.3: Sequence diagram illustrating the workflow of `setup.ps1`.

3.2.3 Inno Setup Installer and Automated Build Scripts

The `installer.iss` script and the automated build script (implemented in `build_installer.py`) together constitute the deployment engine that transforms the prepared package into a polished Windows installer executable. They encapsulate the final integration processes—from system registration and shortcut creation to the orchestration of the complete build workflow—ensuring that the final product is robust, reproducible, and seamlessly integrated with the Windows environment.

Inno Setup Installer Script (`installer.iss`)
Before performing file operations, this script verifies that the staging directory produced by `package_app.py` and refined by `setup.ps1` is complete and correctly structured. It then proceeds to:

- Copy the contents of the staging directory into the final installation folder.
- Create Windows registry entries and shortcuts (for the Start menu and desktop) to ensure smooth system integration.
- Execute the PowerShell setup script (`setup.ps1`) during installation to finalize configuration tasks, such as initializing the embedded Python environment and processing metadata.
- Update the registry to record the installed version (as demonstrated in the `[Code]` section), thereby maintaining accurate version control for future uninstallations.

Automated Build Script (`build_installer.py`)
This Python script orchestrates the entire packaging and build workflow by rigorously validating input parameters and file integrity. It then coordinates the production of the final installer by:

- Utilizing a JSON configuration file (`build_config.json`) as the sole method for specifying critical parameters such as the application root, entry file, application name, version, output directory, and an optional path to `ISCC.exe`.

- Providing a clean configuration approach where developers maintain all build settings in a structured JSON file, eliminating command-line parameter complexity while improving reproducibility and reducing error potential.
- Validating configuration values and ensuring the presence and correctness of essential files (e.g., the application source and the `installer.iss` template).
- Setting sensible defaults for optional parameters (such as version and entry file) when not specified in the configuration file.
- Locating the Inno Setup Compiler (`ISCC.exe`), either by an explicitly provided path in the configuration or by searching standard installation directories.
- Creating a temporary build folder where it stages the entire package—including the application source tree, generated metadata (`metadata.txt`), boot script (`boot.py`), and other internal scripts.
- Invoking the Inno Setup Compiler with parameters passed via `/D` switches (such as `AppName`, `AppVersion`, and `BuildDir`), resulting in a single, ready-to-use installer executable.

Figure 3.4 (shown below) presents an activity diagram illustrating the overall build-and-installation process, from initial packaging on the developer side to final deployment on the user side.

Following all these steps, the build process produces a single, self-contained Windows executable ready to be executed on an end-user's machine.

3.3 Summary

The custom solution presented in this chapter addresses the key limitations identified in both naive approaches and existing tools for Python application distribution. Combining embedded Python interpreters with Windows-native installer technology and PowerShell automation, the architecture balances independence, efficiency, and user experience.

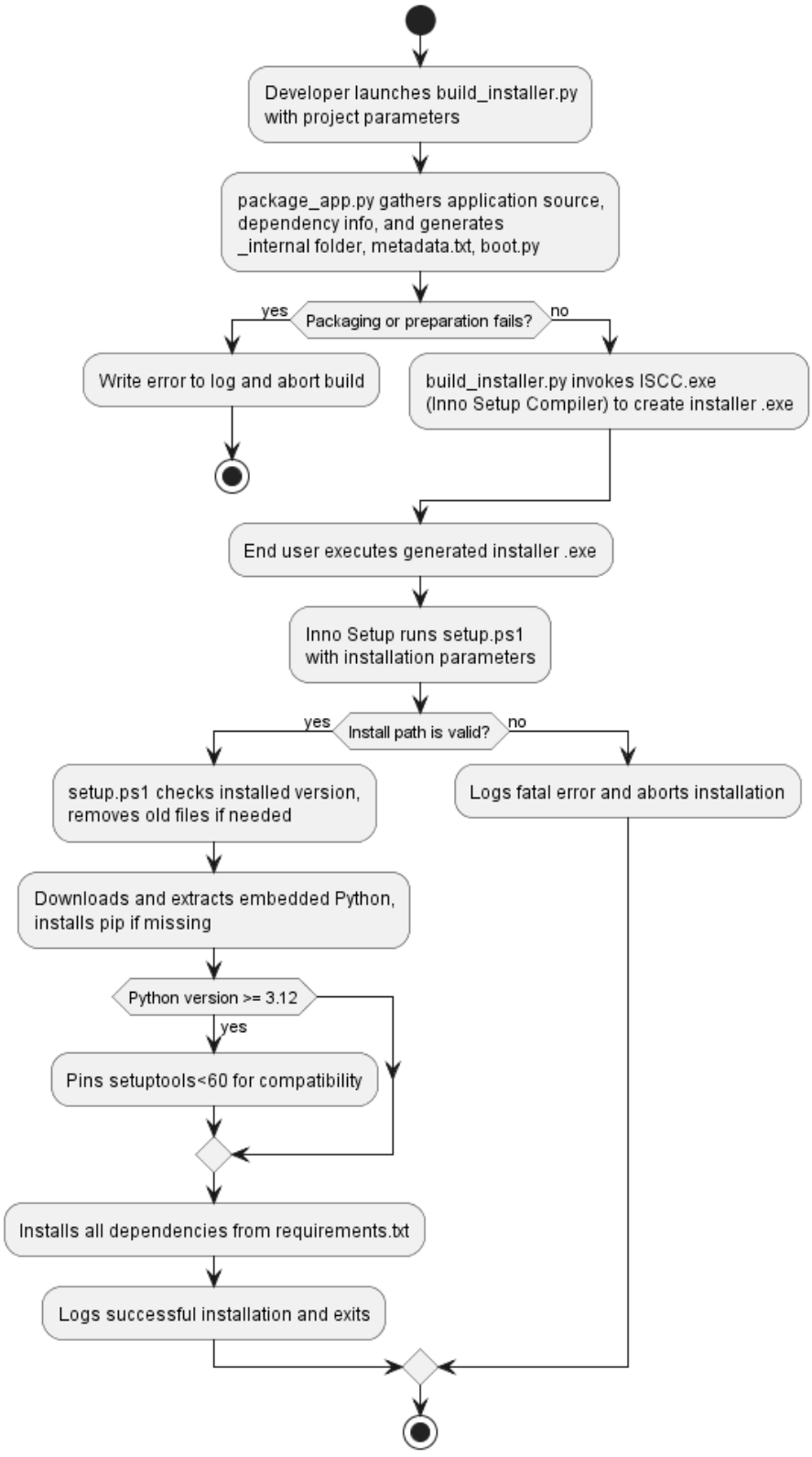


Figure 3.4: Activity diagram of End-to-End Build and Installation Workflow

The solution is structured around four integrated components:

1. **Packaging Script** (`package_app.py`) – Establishes the application structure, filters unnecessary files, and generates essential metadata for the installation process.
2. **Host Setup Script** (`setup.ps1`) – Orchestrates the installation workflow, performing conditional Python environment setup, dependency management, and error handling through a robust sequence of operations.
3. **Inno Setup Configuration** – Transforms the prepared package into a professional Windows installer that integrates with system conventions.
4. **Automated Build Workflow** – Coordinates the process from source code to final installer executable.

This architecture operates entirely in user space, eliminating administrative privilege requirements while maintaining a professional installation experience. Separating the Python interpreter from the application code enables incremental updates with smaller download sizes. The solution prioritizes Windows optimization over cross-platform compatibility, allowing for deeper integration with familiar installation paradigms.

The resulting system provides developers with a streamlined workflow for packaging Python applications that end users can install and update through familiar interfaces, without requiring pre-installed Python environments or technical expertise.

3.4 Key Advantages and Implementation Remarks

The custom PowerShell-based solution provides several strategic advantages that directly address the limitations identified in both naive approaches and existing tools:

- **Balanced Privilege Model:** By operating entirely in user space while maintaining a professional installation experience, the solution overcomes the administrative barriers that disqualify many enterprise deployment options.

- **Incremental Update Efficiency:** The separation of runtime from application code creates a bandwidth-efficient update mechanism that significantly reduces network utilization compared to full reinstallation approaches.
- **Environment Consistency:** The embedded Python approach provides strong isolation without the operational complexity of containerization, preventing the dependency conflicts that commonly plague Python deployments.
- **Streamlined Development Workflow:** The automated build process reduces the specialized knowledge required for packaging, allowing development teams to maintain focus on application functionality.

The presented implementation integrates PowerShell automation, Inno Setup's installer framework, and Python's embeddable package into a cohesive solution. Rather than relying on a single technology, it strategically combines tools to achieve results that surpass what each could offer alone.

Explicitly optimized for Windows, this architecture leverages native platform capabilities and installer conventions to deliver a user experience that mirrors commercial software standards, far beyond the typical Python packaging approach.

By resolving tensions between flexibility, simplicity, and user friendliness, the solution proves that well-integrated technologies can overcome the limitations of conventional Python distribution methods. It demonstrates how thoughtful design leads to a streamlined developer deployment process and a seamless user experience.

4 Comparison of Existing Solutions

The evaluation of Python application distribution methods requires a structured assessment framework that reflects real-world deployment challenges. This chapter compares previously discussed solutions—including naive approaches, existing tools, and the custom PowerShell-based solution—against parameters derived from practical business requirements.

4.1 Evaluation Context and Parameters

The parameters used for evaluation emerged from a collaborative engagement with industry partners who faced significant operational challenges when deploying Python applications in corporate environments. During this collaboration, several critical deployment factors were identified that directly impact the business value and operational feasibility of Python-based tools:

- **Privileges Required:** Corporate security policies often restrict administrative access for standard users, rendering solutions that require elevated permissions impractical for widespread deployment.
- **Third-Party Dependencies:** Solutions requiring additional software components increase security review complexity and may conflict with approved software policies in regulated environments.
- **License Considerations:** Distribution methods with commercial licensing requirements introduce procurement complexities and ongoing cost considerations that impact scalability.
- **Package Size:** Network bandwidth limitations and storage constraints in enterprise environments make package size a significant consideration, particularly for applications deployed across hundreds or thousands of workstations.
- **Installation Complexity:** Applications that present complex or unfamiliar installation interfaces increase support burdens

and reduce user acceptance, particularly in organizations with varying technical expertise.

- **Windows Interface Integration:** Solutions that provide standard Windows interfaces improve user acceptance and reduce training requirements.
- **Update Mechanism:** The business cost of updates—including bandwidth consumption, IT support overhead, and user downtime—directly affects the total cost of ownership for Python applications in production environments.
- **Developer Overhead:** The complexity of maintaining deployment workflows affects development agility and resource requirements.
- **User Space Operation:** Solutions that operate entirely within user directories without requiring system modifications provide crucial flexibility in restricted environments.

These parameters directly reflect the business imperatives that drove the development of the custom solution. Rather than purely technical considerations, they represent the operational realities determining whether a Python application can be effectively deployed and maintained at scale in enterprise settings.

4.2 Detailed Parameter Analysis

Privileges Required

Standard users typically lack administrative access in enterprise environments, making privilege requirements a fundamental deployment consideration. Solutions like PyInstaller, cx_Freeze, and the custom solution operate entirely with standard user permissions, enabling deployment without IT intervention. In contrast, Docker requires administrative privileges to install and configure container services, creating potential adoption barriers in restricted environments [24]. Similarly, Briefcase relies on the WiX toolset, which requires elevated permissions during installation [18]. This single factor can significantly

impact deployment feasibility in environments with strict security policies.

Third-Party Dependencies

Every external dependency introduces potential security vulnerabilities, compatibility issues, and approval requirements in corporate settings. Nuitka requires C++ compilation tools that may not be present on many business workstations [15], while Docker necessitates substantial infrastructure beyond the application itself. PyInstaller and cx_Freeze achieve independence by bundling all dependencies into single executables, though at the cost of larger package sizes. The custom solution emphasizes embedding only essential components while maintaining control over the Python environment, pushing complexity to the developer side rather than the end-user experience.

Licensing Costs

Licensing considerations directly impact scalability and adoption, particularly in large organizations. While most Python distribution methods use free and open-source licenses, Docker Desktop introduced commercial licensing requirements for larger organizations in 2021 [22], potentially creating procurement obstacles. The custom solution's use of freely available components eliminates this barrier. According to JetBrains' 2021 Python Developer Survey, licensing concerns influence tool selection for 31% of enterprises with over 1000 employees [25].

Package Size

Network infrastructure limitations make package size a practical deployment concern rather than a theoretical consideration. Briefcase produces comprehensive but extensive packages by bundling complete runtime environments [18]. Similarly, Docker images containing Python environments typically require substantial storage. The custom solution aims to optimize for bandwidth-constrained environments by separating the Python runtime from application code and enabling delta updates for application components.

Installation Complexity

User experience during installation directly affects adoption rates and support requirements. Manual setup approaches create high friction by requiring users to follow technical instructions, while Docker introduces container concepts unfamiliar to most business users. The BeeWare project recognizes this challenge by emphasizing native installation experiences [17], as does the custom solution with its standard Windows installer interface. Installation complexity influences user perception of application quality and reliability throughout the application lifecycle.

Update Mechanism

The ability to efficiently distribute updates determines long-term operational costs and application reliability. Most Python packaging tools lack integrated update mechanisms, requiring complete reinstallation for minor changes. PyUpdater initially addressed this gap with incremental updates but has since ended official support [11], creating sustainability risks. The custom solution implements a delta update approach that aims to minimize network impact while maintaining version control, allowing security patches and bug fixes to be deployed without requiring complete reinstallation.

Developer Overhead

Complex deployment tooling diverts development resources from core application functionality. Nuitka's compilation approach requires additional build steps and testing cycles [16], while Docker demands container management expertise. PyInstaller and cx_Freeze achieve relatively streamlined workflows but lack enterprise deployment features. The custom solution automates packaging and distribution processes through PowerShell, enabling developers to focus primarily on application logic rather than deployment mechanics.

Environment Isolation

Python applications frequently encounter "dependency hell" when different components require incompatible package versions. Docker

provides maximum isolation through containerization [21], while PyInstaller and cx_Freeze bundle dependencies into self-contained executables. Nuitka's compilation approach reduces but doesn't eliminate environmental dependencies [16]. The custom solution achieves isolation by embedding a dedicated Python interpreter and managing package installations within a controlled environment. According to GitHub's 2021 Octoverse report, dependency conflicts remain among the top challenges for Python developers [26].

4.3 Competitive Positioning

While each solution offers specific advantages for particular deployment scenarios, the comparative analysis demonstrates why existing tools did not fully address the partner organization's requirements for Python application distribution:

- **PyInstaller and cx_Freeze:** While offering simplicity and independence from pre-installed Python, these solutions lack efficient update mechanisms—a critical requirement for applications that evolve frequently in response to business needs.
- **PyUpdater:** Despite its update capabilities, PyUpdater's discontinued official support raised sustainability concerns for long-term enterprise deployment, and its approach still produces relatively large initial packages.
- **Chocolatey:** The administrative privilege requirement formed an insurmountable barrier in security-restricted environments, despite Chocolatey's strong Windows integration and package management capabilities.
- **Briefcase:** While providing excellent native integration, Briefcase's large package size and reinstallation-based updates proved inefficient for deployments across bandwidth-constrained networks.
- **Docker:** The container-based approach introduced excessive complexity for desktop applications and required specialized knowledge that could not be assumed across the user base.

The custom PowerShell-based solution was designed to occupy the previously unfilled niche at the intersection of user-space operation, incremental update capability, and familiar Windows integration. This positioning directly addresses the operational constraints identified during the industry collaboration, enabling Python applications to be deployed with greater efficiency and a lower total cost of ownership than existing alternatives.

4.4 Comparative Analysis

The following tables comprehensively compare the Python distribution solutions evaluated in this thesis, focusing specifically on Windows deployment scenarios.

Table 4.1: Installation Requirements

Solution	Privileges Required	Dependencies	License
Manual Setup	Standard	Python interpreter	Free
PyInstaller	Standard	None	Free
PyUpdater	Standard	None	Free
Chocolatey	Admin	Chocolatey	Free/Paid
cx_Freeze	Standard	None	Free
Nuitka	Standard	C++ toolchain	Free
Briefcase	Admin	WiX Toolset	Free
Docker	Admin	Docker Engine	Free/Paid
Custom Solution	Standard	None*	Free

*Inno Setup is required, but only on the developer side, not on end-user systems

Table 4.2: End-User Experience

Solution	Package Size	Installation Complexity	Native Windows interface
Manual Setup	Minimal	High	No
PyInstaller	Large	Low	No
PyUpdater	Large	Low	No
Chocolatey	Medium	Medium	No
cx_Freeze	Large	Medium	No
Nuitka	Medium	Low	No
Briefcase	Very Large	Low	Yes
Docker	Very Large	High	No
Custom Solution	Low	Low	Yes

Table 4.3: Maintenance and Operations

Solution	Updates	Dev Overhead	Environment Isolation
Manual Setup	None	Medium	Low
PyInstaller	None	Low	High
PyUpdater	Incremental	High	High
Chocolatey	Full	Medium	Low
cx_Freeze	None	Low	High
Nuitka	None	High	High
Briefcase	Full	Medium	High
Docker	Layer-based	High	Very High
Custom Solution	Incremental	Low	High

5 Future Work

While the developed solution successfully addresses the core requirements for Python application distribution in Windows environments, several opportunities for enhancement can be identified. The following areas represent natural extensions of the current work that could further improve the solution's utility and adoption:

Digital Signing and Security

The current implementation focuses on functional deployment, but modern software distribution increasingly demands robust security measures. Implementing digital signatures for installers and update packages would enhance trust and verify package integrity. This could be achieved through integration with Windows certificate store and standard code signing practices.

Rollback and Recovery Mechanisms

The implementation sequence diagram (Figure 3.3) shows that the update process follows a linear path. Adding rollback capabilities would enhance reliability by preserving system state before updates and providing automatic recovery options if installation fails. This would be particularly valuable in environments where application availability is critical.

Customizable Installer Appearance

While Inno Setup provides a professional installation experience, organizations often require branded deployment tools. Extending the solution to support customizable themes, logos, and styling would enable better integration with corporate identity requirements without compromising the underlying functionality.

Graphical User Interface for Developers

The current command-line interface efficiently serves technical users, but a graphical tool for managing deployments could broaden adoption. Such an interface could simplify configuration management, dependency selection, and update deployment while maintaining the solution's core automation capabilities.

6 Conclusion

This thesis aimed to design and implement a self-contained distribution solution for Python applications in Windows environments. The initial investigation identified significant limitations in naive deployment approaches, prompting a comprehensive analysis of existing tools and frameworks in the Python ecosystem.

This systematic examination, detailed in Chapter 2, revealed that while individual solutions excel in specific deployment aspects, none adequately addressed the requirements necessary for seamless distribution. PyInstaller offers simplicity but lacks update mechanisms; Docker provides strong isolation at the cost of complexity; Briefcase delivers professional installation experiences but requires administrative privileges. The comparative analysis presented in Tables 4.1, 4.2, and 4.3 quantifies these tradeoffs across critical parameters including installation requirements, user experience factors, and maintenance considerations.

Drawing upon these findings, a novel PowerShell-based solution was developed that strategically integrates the strengths of multiple technologies. By combining Inno Setup's installer with PowerShell's automation capabilities and Python's embeddable package, the solution achieves a synthesis that overcomes the limitations of individual approaches. This architecture manages the complete deployment lifecycle from initial installation through dependency management to configuration, all while maintaining user-space operation.

The comparative analysis demonstrates that this custom solution effectively addresses the identified shortcomings of existing tools. It operates without administrative privileges, provides a familiar installation experience, achieves strong environment isolation, and supports delta updates. This combination of capabilities significantly contributes to Python application distribution for Windows environments.

By integrating proven technologies into a coherent deployment framework, this thesis demonstrates that careful architectural design can reconcile seemingly contradictory requirements. The solution allows developers to focus on application functionality rather than distribution mechanics, removing a significant barrier to Python adoption in complex computing environments.

Bibliography

1. ALSALEH, Tarek. *An Approach to Distributing Python Apps* [online]. Medium (Retail Insight blog), 2022-02-09 [visited on 2024-05-15]. Available from: `https://medium.com/retail-insight/an-approach-to-distributing-python-apps-d9904b2d5916`.
2. LEE, Luke. Using PyInstaller to Easily Distribute Python Applications. *Real Python* [online]. 2019 [visited on 2024-05-15]. Available from: `https://realpython.com/pyinstaller-python/`.
3. COLLICUTT, Curtis. *Python Program Distribution and Installation Methods* [online]. Server as Code, 2024-12-21 [visited on 2025-04-15]. Available from: `https://serverascode.com/2024/12/21/python-installation-methods.html`.
4. ALSALEH, Tarek. *Distributing Python applications* [online]. Retail Insight Tech Blog, 2022 [visited on 2025-04-15]. Available from: `https://www.retailinsight.io/blog/distributing-python-applications`. Discusses deployment challenges.
5. *Installing Python Modules* [online]. Python Software Foundation, 2023. [visited on 2025-04-15]. Available from: `https://docs.python.org/3/installing/`. Python v3.x Manual.
6. ZACZYŃSKI, Bartosz. Install and Execute Python Applications Using pipx. *Real Python* [online]. 2020 [visited on 2025-04-15]. Available from: `https://realpython.com/python-pipx/`.
7. COLLICUTT, Curtis. *Installer Scripts vs. Modern Python Distribution Methods* [online]. Server as Code blog, 2024-12-21 [visited on 2025-04-15]. Available from: `https://serverascode.com/2024/12/21/python-installation-methods.html`. Emphasizes pros/cons of curl|bash installers.
8. PYTHON PACKAGING AUTHORITY. *Overview of Python Packaging – Targeting a Non-Developer Audience* [online]. Python Packaging User Guide, 2023 [visited on 2025-04-17]. Available from: `https://packaging.python.org/overview/`.
9. PYINSTALLER DEVELOPMENT TEAM. *PyInstaller Manual* [online]. PyInstaller Development Team, 2025. [visited on 2025-05-12]. Available from: `https://pyinstaller.org/`.

10. PYUPDATER DEVELOPERS. *PyUpdater Official Documentation* [online]. 2025. [visited on 2025-05-12]. Available from: https://www.pyupdater.org/.

11. JMSWAG. *PyUpdater - Official Support Ended* [online]. 2022-12. [visited on 2024-05-15]. Available from: https://github.com/Digital-Sapphire/PyUpdater.

12. PYUPDATER2 CONTRIBUTORS. *PyUpdater2 – A Community Fork of PyUpdater* [online]. 2025. [visited on 2025-05-12]. Available from: https://github.com/pyupdater2/pyupdater2.

13. *Chocolatey Documentation* [online]. Chocolatey Software, Inc., 2025. [visited on 2025-05-12]. Available from: https://chocolatey.org/docs.

14. CX_FREEZE DEVELOPERS. *cx_Freeze Documentation* [online]. 2025. [visited on 2025-05-12]. Available from: https://cx-freeze.readthedocs.io/.

15. NUITKA PROJECT. *Nuitka User Manual* [online]. 2025. [visited on 2025-05-12]. Available from: https://nuitka.net/doc/user-manual.html.

16. HAYEN, Kay. *Nuitka Performance - Compiled vs Interpreted Python* [online]. 2023. [visited on 2024-05-16]. Available from: https://nuitka.net/pages/performance.html.

17. KEITH-MAGEE, Russell. *The BeeWare Project* [online]. 2023. [visited on 2024-05-16]. Available from: https://beeware.org/.

18. BEEWARE PROJECT. *BeeWare Briefcase Documentation* [online]. 2025. [visited on 2025-05-12]. Available from: https://briefcase.readthedocs.io/en/latest/.

19. BEEWARE PROJECT. *The BeeWare Suite of Tools and Libraries* [online]. 2023. [visited on 2024-05-16]. Available from: https://beeware.org/project/.

20. DOCKER, INC. *Docker Documentation* [online]. Docker, Inc., 2025. [visited on 2025-05-12]. Available from: https://docs.docker.com/.

21. DOCKER, INC. *Best practices for writing Dockerfiles for Python applications* [online]. Docker, Inc., 2024. [visited on 2025-05-12]. Available from: `https://docs.docker.com/language/python/build-images/`.

22. DOCKER, INC. *Docker Engine Source Code* [online]. Docker, Inc., 2025 [visited on 2025-05-12]. Available from: `https://github.com/moby/moby`.

23. DOCKER, INC. *Docker Subscription Service Agreement* [online]. Docker, Inc., 2024 [visited on 2025-05-12]. Available from: `https://www.docker.com/legal/docker-subscription-service-agreement/`.

24. MICROSOFT DEVELOPER BLOG. Windows Containers and Python: Limitations and Workarounds. *Microsoft Developer Blog* [online]. 2024 [visited on 2025-05-12]. Available from: `https://devblogs.microsoft.com/python/`.

25. JETBRAINS. *Python Developers Survey 2021* [online]. 2021. [visited on 2022-03-15]. Available from: `https://www.jetbrains.com/lp/python-developers-survey-2021/`.

26. GITHUB. *The State of the Octoverse 2021* [online]. 2021. [visited on 2022-02-10]. Available from: `https://octoverse.github.com/2021/`.

A Appendix

A.1 Source Code Repository

The complete source code for the Python distribution solution described in this thesis is available in the following GitHub repository:

`https://github.com/Ricardo1356/Distributing-standalone-Python-applications`

This repository contains all components of the distribution system, including:

- **Build System:** The `build_installer.py` script that processes configuration from `build_config.json`, `package_app.py` for collectionn and preparation application files, and the Inno Setup script template.
- **Installation Components:** The PowerShell script (`setup.ps1`)is responsible for setting up the Python environment, and the boot mechanism via `boot.py`.
- **Example Application:** A sample application providing a working example to be distributed using the packaging system.
- **Documentation:** UML diagrams and detailed README file.

The repository is structured to allow easy exploration of the implementation details. Additionally, a complete snapshot of this repository is included in the electronic attachments accompanying this thesis submission.