PowerShell 7.x Fundamentals

Modern Cross-Platform Automation and Scripting with PowerShell

Preface

When I first began working with PowerShell, it was a Windows-only tool — powerful, certainly, but confined to a single ecosystem. Today, PowerShell 7.x stands as something far more ambitious: a modern, cross-platform automation engine that runs on Windows, macOS, and Linux alike. That transformation is not just a technical achievement; it represents a fundamental shift in how IT professionals, developers, and system administrators approach their daily work. This book was written to help you navigate that shift with confidence.

Why This Book Exists

PowerShell 7.x Fundamentals was born from a simple observation: too many people still think of PowerShell as "just a Windows command prompt with extra steps." Nothing could be further from the truth. PowerShell is a fully realized scripting language built on .NET, capable of managing cloud infrastructure, automating complex workflows, parsing APIs, and orchestrating tasks across heterogeneous environments. Yet for all its power, getting started with PowerShell — *really* getting started, with a solid foundation — can feel daunting.

This book aims to change that. Whether you are a complete beginner who has never opened a PowerShell console or an IT professional looking to formalize and deepen your existing PowerShell knowledge, these pages will give you a structured, practical path from first principles to real-world automation.

What You Will Learn

The book is organized into a deliberate progression. We begin with the essentials — understanding *what* PowerShell 7.x is and how to install and run it across platforms (**Chapters 1–2**). From there, we dive into the core language mechanics: cmdlets and the object pipeline, variables, data types, operators, and control flow (**Chapters 3–6**). These chapters form the backbone of everything you will do in PowerShell.

The middle section moves into applied territory. You will learn to work with the file system, manage processes and services, and interact with the operating system directly through PowerShell (**Chapters 7–8**). Then we turn to the craft of writing reusable, maintainable PowerShell code — functions, scripts, error handling, and modules (**Chapters 9–12**).

The final chapters bring it all together. You will build real automation workflows, work with data formats like JSON, CSV, and XML, consume REST APIs, and adopt PowerShell best practices that distinguish professional-quality scripts from quick-and-dirty one-liners (**Chapters 13–15**). **Chapter 16** looks ahead, bridging the fundamentals covered here with the advanced PowerShell techniques you will encounter next in your journey.

Five appendices — including a cmdlet cheat sheet, common error reference, pipeline guide, example automation scripts, and a learning roadmap — ensure this book remains a useful companion long after you finish reading it.

How to Use This Book

If you are new to PowerShell, I encourage you to read the chapters in order. Each one builds on the last. If you already have PowerShell experience, feel free to jump

to the topics that matter most to you — the chapters are designed to be self-contained enough to serve as reference material.

Throughout the book, I have prioritized *clarity over cleverness*. Every concept is explained in plain language, reinforced with practical examples, and connected to scenarios you are likely to encounter in the real world. PowerShell is a tool meant to make your life easier, and learning it should feel the same way.

Acknowledgments

No book is written in isolation. I owe a debt of gratitude to the PowerShell open-source community, whose tireless contributions have made PowerShell 7.x what it is today. I am also deeply thankful to the technical reviewers who challenged my assumptions, the editors who sharpened my prose, and the readers of early drafts whose feedback shaped every chapter for the better.

Most of all, thank you — for picking up this book and investing your time in learning PowerShell. The skills you build here will pay dividends across your entire career.

Let's get started.

Laszlo Bocso (MCT), 2026

Table of Contents

Chapter	Title	Page
1	What PowerShell 7.x Really Is	6
2	Installing and Running PowerShell 7.x	20
3	Cmdlets and the Object Pipeline	37
4	Variables, Data Types, and Objects	52
5	Operators and Expressions	70
6	Conditional Logic and Loops	90
7	Files, Directories, and the File System	112
8	Processes, Services, and System Interaction	129
9	Writing Functions	150
10	Writing PowerShell Scripts	173
11	Error Handling and Exceptions	191
12	Working with Modules	213
13	Automating Tasks with PowerShell	230
14	Working with Data Formats and APIs	257
15	PowerShell Best Practices	275
16	From PowerShell Fundamentals to Advanced Automation	294
App	Essential PowerShell Cmdlet Cheat Sheet	312
App	Common PowerShell Errors and Fixes	335
App	Object Pipeline Reference	351
App	Example Automation Scripts	367
App	PowerShell Learning Roadmap	392

Chapter 1: What PowerShell 7.x Really Is

Introduction

Welcome to the very first chapter of this book. Before you write a single line of PowerShell code, before you automate your first task, and before you build your first script, it is essential that you understand what PowerShell truly is at its core. This chapter is not about memorizing commands or syntax. It is about building a solid mental model of the tool you are about to spend a great deal of time mastering. Too many professionals jump straight into writing scripts without understanding the philosophy, architecture, and evolution behind PowerShell. That approach leads to fragile scripts, misunderstandings about how the shell works, and frustration when things do not behave as expected. This chapter will ensure you start on the right foundation.

The Evolution of PowerShell: From Monad to PowerShell 7.x

The story of PowerShell begins long before the name "PowerShell" ever existed. In the early 2000s, a Microsoft engineer named Jeffrey Snover recognized a fundamental problem with Windows system administration. While Linux and Unix admin-

istrators had powerful text-based shells like Bash, Ksh, and Zsh that allowed them to pipe text between commands and automate complex workflows, Windows administrators were stuck with two limited tools: the legacy Command Prompt (cmd.exe) and Windows Script Host (which ran VBScript or JScript). Neither of these tools was designed for the kind of structured, scalable automation that enterprise environments demanded.

Snover authored a visionary document called the "Monad Manifesto" in 2002. In this document, he proposed a new kind of shell, one that would not pass plain text between commands, but instead would pass structured .NET objects. This single design decision would become the defining characteristic that separates PowerShell from every other shell in existence. The Monad project evolved through several beta releases before being officially released as Windows PowerShell 1.0 in November 2006.

Over the following decade, Windows PowerShell matured through several major versions:

Version	Release Year	Key Milestone
Windows PowerShell 1.0	2006	Initial release with 129 cmdlets, basic pipeline, and remoting concepts
Windows PowerShell 2.0	2009	Introduced PowerShell Remoting (WinRM), background jobs, modules, and the ISE editor
Windows PowerShell 3.0	2012	Workflows, scheduled jobs, improved module auto-loading, CIM cmdlets
Windows PowerShell 4.0	2013	Desired State Configuration (DSC), enhanced debugging
Windows PowerShell 5.0	2016	PowerShell classes, PowerShellGet and the PowerShell Gallery, OneGet package management

Windows PowerShell 5.1	2017	Final release of Windows PowerShell, shipped with Windows 10 and Windows Server 2016
PowerShell Core 6.0	2018	First cross-platform release built on .NET Core, open source on GitHub
PowerShell Core 6.1/6.2	2018-2019	Compatibility improvements, performance enhancements
PowerShell 7.0	2020	Dropped "Core" from the name, unified experience, parallel ForEach-Object, ternary operator
PowerShell 7.1 through 7.4	2020-2024	Incremental improvements, new operators, improved error handling, .NET updates
PowerShell 7.5	2025	Built on .NET 9, enhanced security features, improved tab completion, native command error handling

The transition from Windows PowerShell 5.1 to PowerShell 7.x represents the most significant architectural shift in PowerShell's history. Windows PowerShell was built on the full .NET Framework, which meant it could only run on Windows. PowerShell 7.x is built on .NET 7, 8, and 9 (depending on the specific release), which is a cross-platform runtime. This means that the same PowerShell scripts you write on Windows can, in many cases, run on Linux and macOS without modification.

Note: Windows PowerShell 5.1 is not deprecated. It ships with every modern version of Windows and will continue to be supported. However, it will not receive new features. All new development, new cmdlets, new language features, and performance improvements happen exclusively in PowerShell 7.x. Throughout this book, when we say "PowerShell," we mean PowerShell 7.x unless explicitly stated otherwise.

Understanding What PowerShell Actually Is

One of the most common misconceptions about PowerShell is that it is simply a command-line interface, a replacement for the Command Prompt on Windows. While PowerShell does provide a command-line interface, describing it this way is like describing a smartphone as "a device that makes phone calls." It is technically accurate but misses almost everything that makes it powerful.

PowerShell is three things simultaneously:

First, PowerShell is a command-line shell. You can open a PowerShell terminal and type commands interactively, just as you would in Bash on Linux or cmd.exe on Windows. You can navigate the file system, run programs, and see output on screen. This is the most visible aspect of PowerShell and the one most people encounter first.

Second, PowerShell is a scripting language. You can write `.ps1` files that contain sequences of commands, logic, loops, functions, error handling, and more. PowerShell scripts can be as simple as a three-line file that restarts a service, or as complex as a multi-thousand-line application that provisions entire cloud environments. The scripting language supports variables, arrays, hashtables, conditional statements, loops, functions, classes, exception handling, and much more.

Third, PowerShell is an automation framework. This is the aspect that most beginners overlook but that experienced professionals consider the most important. PowerShell provides a standardized way to manage and automate virtually any technology. Through its module system, PowerShell can be extended to manage Active Directory, Azure, AWS, VMware, Exchange, SQL Server, Docker, Kubernetes, and hundreds of other technologies. Each of these modules provides cmdlets (pronounced "command-lets") that follow consistent naming conventions and behavioral patterns.

Let us look at a simple example that demonstrates all three aspects working together:

```
# Interactive shell usage: check running processes
Get-Process | Where-Object { $_.CPU -gt 100 } | Sort-Object CPU
-Descending

# Scripting: save the above as a reusable script with parameters
param(
    [int]$CpuThreshold = 100
)

$heavyProcesses = Get-Process | Where-Object { $_.CPU -gt
$CpuThreshold }
$heavyProcesses | Sort-Object CPU -Descending | Format-Table
Name, Id, CPU -AutoSize

# Automation framework: extend to remote management
Invoke-Command -ComputerName "Server01", "Server02", "Server03"
-ScriptBlock {
    Get-Process | Where-Object { $_.CPU -gt 100 } | Sort-Object
CPU -Descending
}
```

In the example above, the same fundamental concept (finding processes with high CPU usage) scales from a single interactive command to a multi-server automation task. This scalability is by design and is one of PowerShell's greatest strengths.

The Object Pipeline: PowerShell's Defining Feature

If there is one concept you must understand deeply to use PowerShell effectively, it is the object pipeline. This is what makes PowerShell fundamentally different from Bash, Zsh, cmd.exe, and every other traditional shell.

In a traditional shell like Bash, when you pipe the output of one command to another, you are passing text. The receiving command must parse that text, figure out which parts are meaningful, and extract the data it needs. This approach is fragile. If the output format changes even slightly (an extra space, a different column order, a localized date format), your text parsing breaks.

In PowerShell, when you pipe the output of one cmdlet to another, you are passing .NET objects. Each object has properties (data) and methods (actions). The receiving cmdlet can access any property of the object directly by name, without any text parsing whatsoever.

Consider this comparison:

In Bash (text pipeline):

```
# Get process information and extract the name of processes using
more than 100MB
ps aux | awk '$6 > 102400 {print $11}'
```

This command depends on the exact column positions in the `ps` output. If the output format changes across different Linux distributions or versions, the `awk` command may extract the wrong column.

In PowerShell (object pipeline):

```
# Get process information and extract the name of processes using
more than 100MB
Get-Process | Where-Object { $_.WorkingSet64 -gt 100MB } |
Select-Object Name
```

This command accesses the `WorkingSet64` property by name. It does not matter how the output is formatted on screen. The property name is a stable contract provided by the .NET object. Notice also that PowerShell understands `100MB` as a numeric value (104,857,600 bytes). This kind of built-in intelligence permeates the entire language.

Let us examine what an object looks like in practice:

```powershell
# Get a single process and examine its properties
$process = Get-Process -Name "pwsh" | Select-Object -First 1

# View all properties
$process | Get-Member -MemberType Property
```

The `Get-Member` cmdlet is one of the most important discovery tools in PowerShell. It shows you all the properties and methods available on any object. When you run the command above, you will see output similar to this:

Name	MemberType	Definition
BasePriority	Property	int BasePriority {get;}
CPU	Property	double CPU {get;}
Handle	Property	System.IntPtr Handle {get;}
HandleCount	Property	int HandleCount {get;}
Id	Property	int Id {get;}
MachineName	Property	string MachineName {get;}
MainWindowTitle	Property	string MainWindowTitle {get;}
Name	Property	string Name {get;}
Path	Property	string Path {get;}
ProcessName	Property	string ProcessName {get;}
StartTime	Property	datetime StartTime {get;}
WorkingSet64	Property	long WorkingSet64 {get;}

Every single one of these properties is accessible in the pipeline. You can filter on any of them, sort by any of them, select any combination of them, and export them to CSV, JSON, XML, or any other format. This is the power of the object pipeline.

PowerShell 7.x Architecture

Understanding the architecture of PowerShell 7.x helps you understand why it behaves the way it does and how to troubleshoot issues when they arise.

At the highest level, PowerShell 7.x consists of several layers:

Layer	Component	Purpose
Runtime	.NET 9 (for PowerShell 7.5)	Provides the underlying runtime environment, garbage collection, type system, and standard libraries
Engine	System.Management.Automation	The core PowerShell engine that handles parsing, pipeline execution, command discovery, and session management
Host	pwsh.exe / ConsoleHost	The application that hosts the PowerShell engine and provides the user interface (console, colors, input/output)
Modules	Built-in and external modules	Collections of cmdlets, functions, and scripts that extend PowerShell's capabilities
Providers	FileSystem, Registry, Variable, etc.	Adapters that expose different data stores through a consistent file-system-like interface

One of the most elegant aspects of PowerShell's architecture is the provider system. PowerShell providers allow you to navigate different data stores using the same commands you use for the file system. For example:

```
# Navigate the file system
Set-Location C:\Users
Get-ChildItem

# Navigate the Windows Registry (Windows only)
Set-Location HKLM:\SOFTWARE
Get-ChildItem
```

```
# Navigate environment variables
Set-Location Env:
Get-ChildItem

# Navigate PowerShell variables
Set-Location Variable:
Get-ChildItem
```

In every case above, `Set-Location` and `Get-ChildItem` work identically. The provider system abstracts the underlying data store and presents it through a consistent interface. This is a powerful example of PowerShell's design philosophy: learn a concept once, apply it everywhere.

Cross-Platform Reality in 2026

PowerShell 7.x runs on Windows, Linux, and macOS. This is not a theoretical capability or a marketing claim. It is a practical reality that thousands of organizations rely on daily. However, it is important to understand what "cross-platform" means and what it does not mean.

PowerShell itself, the language, the engine, the pipeline, the module system, all of these work identically across all platforms. A `ForEach-Object` loop works the same on Ubuntu as it does on Windows. A hashtable is a hashtable regardless of the operating system. String manipulation, regular expressions, error handling, and all other language features are platform-independent.

What does vary across platforms is the availability of certain modules and cmdlets. For example:

```
# This works on all platforms
Get-Process
Get-ChildItem
Invoke-RestMethod
```

```
ConvertTo-Json

# This works only on Windows
Get-Service
Get-EventLog
Get-WmiObject

# This works on Linux and macOS (through native commands)
pwsh -Command "& { /usr/bin/uname -a }"
```

Note: When a cmdlet is not available on your platform, PowerShell will give you a clear error message. You will not encounter silent failures or mysterious behavior. PowerShell 7.x has invested heavily in making cross-platform differences explicit and understandable.

To check your current PowerShell version and platform, use the built-in `$PSVersionTable` automatic variable:

```
$PSVersionTable
```

This will produce output similar to:

```
Name                           Value
----                           -----
PSVersion                      7.5.0
PSEdition                      Core
GitCommitId                    7.5.0
OS                             Microsoft Windows 10.0.22631
Platform                       Win32NT
PSCompatibleVersions           {1.0, 2.0, 3.0, 4.0, 5.0, 5.1,
6.0, 6.1, 6.2, 7.0, 7.1, 7.2, 7.3, 7.4, 7.5}
PSRemotingProtocolVersion      2.3
SerializationVersion           1.1.0.1
WSManStackVersion              3.0
```

The `PSEdition` value of `Core` tells you that you are running PowerShell 7.x (built on .NET Core / .NET). If you see `Desktop`, you are running the legacy Windows PowerShell 5.1.

The Verb-Noun Naming Convention

Every cmdlet in PowerShell follows a strict `Verb-Noun` naming convention. This is not merely a suggestion or a best practice. It is an enforced design pattern that makes PowerShell remarkably discoverable and consistent.

The verb describes the action. The noun describes what the action operates on. Here are some examples:

Cmdlet	Verb	Noun	What It Does
Get-Process	Get	Process	Retrieves running processes
Stop-Process	Stop	Process	Terminates a running process
Get-Service	Get	Service	Retrieves Windows services
Start-Service	Start	Service	Starts a stopped Windows service
Get-Content	Get	Content	Reads the content of a file
Set-Content	Set	Content	Writes content to a file
New-Item	New	Item	Creates a new file, directory, or other item
Remove-Item	Remove	Item	Deletes a file, directory, or other item
Invoke-RestMethod	Invoke	RestMethod	Sends an HTTP request to a REST API
ConvertTo-Json	ConvertTo	Json	Converts an object to JSON format

PowerShell defines a set of approved verbs that all cmdlets should use. You can see the complete list by running:

```
Get-Verb | Sort-Object Verb | Format-Table Verb, Group,
Description -AutoSize
```

This naming convention means that once you learn the pattern, you can often guess the name of a cmdlet you have never used before. If you know `Get-Process` exists, you can reasonably guess that `Stop-Process`, `Start-Process`,

and `Wait-Process` also exist. If you know `Get-Service` exists, you can guess `Stop-Service`, `Start-Service`, and `Restart-Service`.

Practical Exercise: Your First PowerShell Exploration

Now that you understand what PowerShell is, let us put that understanding into practice. Open a PowerShell 7.x terminal and work through the following exercises. Do not just read them. Type each command and observe the output carefully.

Exercise 1: Verify Your Environment

```
# Check your PowerShell version
$PSVersionTable

# Check your current location in the file system
Get-Location

# Check the current date and time (returned as a DateTime object)
Get-Date

# Examine the object returned by Get-Date
Get-Date | Get-Member
```

Exercise 2: Explore the Object Pipeline

```
# Get all running processes
Get-Process

# Filter to only processes using more than 50MB of memory
Get-Process | Where-Object { $_.WorkingSet64 -gt 50MB }

# Select only the Name and WorkingSet64 properties, sorted by
memory usage
Get-Process | Where-Object { $_.WorkingSet64 -gt 50MB } |
    Sort-Object WorkingSet64 -Descending |
```

```
    Select-Object Name,
@{Name='MemoryMB';Expression={[math]::Round($_.WorkingSet64 /
1MB, 2)}}
```

Exercise 3: Discover Available Commands

```
# Count how many commands are available
(Get-Command).Count

# Find all commands with the verb "Get"
Get-Command -Verb Get | Measure-Object

# Find all commands related to "Process"
Get-Command -Noun Process

# Get help for a specific cmdlet
Get-Help Get-Process -Detailed
```

Exercise 4: Experience the Provider System

```
# List all available PowerShell providers
Get-PSProvider

# List all available PowerShell drives
Get-PSDrive

# Navigate to the environment variable provider
Set-Location Env:
Get-ChildItem

# Return to the file system
Set-Location ~
```

After completing these exercises, you should have a tangible sense of how PowerShell works. You have seen objects in the pipeline, used the Verb-Noun naming convention, explored the provider system, and used the built-in discovery tools (`Get-Command`, `Get-Help`, `Get-Member`) that will be your constant companions throughout this book.

Summary

PowerShell 7.x is far more than a command-line shell. It is a complete automation platform built on the .NET runtime that combines an interactive shell, a full-featured scripting language, and an extensible automation framework into a single, cohesive tool. Its object-based pipeline eliminates the fragility of text parsing that plagues traditional shells. Its cross-platform support means you can use the same tool and the same skills whether you are managing Windows servers, Linux containers, or macOS development machines. Its consistent Verb-Noun naming convention and rich discovery tools make it one of the most learnable and approachable programming environments available today.

In the chapters that follow, we will build on this foundation systematically. You will learn the language syntax, master the pipeline, write robust scripts, handle errors gracefully, work with modules, manage remote systems, and ultimately become confident in using PowerShell to automate anything your work demands. But everything starts here, with a clear understanding of what PowerShell 7.x really is and why it was designed the way it was.

Chapter 2: Installing and Running PowerShell 7.x

Introduction

Before you can harness the full power of PowerShell 7.x for automation, scripting, and system administration, you need to install it properly on your operating system and understand how to launch and configure it effectively. This chapter walks you through every step of that process, from understanding the difference between Windows PowerShell and PowerShell 7.x to installing it on Windows, macOS, and Linux. By the end of this chapter, you will have a fully functional PowerShell 7.x environment ready for the exercises and concepts explored throughout the rest of this book.

One of the most common sources of confusion for newcomers and even experienced administrators is the relationship between Windows PowerShell (version 5.1) and PowerShell 7.x. It is essential to understand that these are two distinct products. Windows PowerShell 5.1 ships with Windows 10 and Windows 11 and is built on the .NET Framework. PowerShell 7.x, on the other hand, is built on .NET 8 (and later .NET 9), is open source, and runs on Windows, macOS, and Linux. They can coexist on the same machine without conflict, and in many enterprise environments, both are installed side by side.

2.1 Understanding the PowerShell Landscape

Before installing anything, it is worth taking a moment to understand the broader landscape of PowerShell versions and why PowerShell 7.x represents the future of the platform.

Aspect	Windows PowerShell 5.1	PowerShell 7.x
.NET Runtime	.NET Framework 4.x	.NET 8 / .NET 9
Open Source	No	Yes (MIT License)
Cross-Platform	No (Windows only)	Yes (Windows, macOS, Linux)
Executable Name	powershell.exe	pwsh.exe (or pwsh on Linux/macOS)
Default Install Location (Windows)	C:\Windows\System32\WindowsPowerShell\v1.0	C:\Program Files\PowerShell\7
Module Compatibility	Full Windows module support	Most modules supported; some Windows-only modules may not work
Active Development	No (maintenance mode only)	Yes (actively developed by Microsoft)
Configuration File	profile.ps1 in WindowsPowerShell folder	profile.ps1 in PowerShell folder
Package Manager Support	Limited	Extensive (winget, brew, apt, snap, and more)

Microsoft has officially stated that Windows PowerShell 5.1 will receive only security fixes and critical bug patches. All new features, performance improvements, and language enhancements are being delivered exclusively through PowerShell 7.x. This means that if you want access to features like pipeline parallelization with

`ForEach-Object -Parallel`, ternary operators, null-coalescing operators, and improved error handling, PowerShell 7.x is where you need to be.

Note: Throughout this book, when we refer to "PowerShell" without a version qualifier, we mean PowerShell 7.x. When we specifically mean the legacy version, we will always say "Windows PowerShell 5.1."

2.2 Installing PowerShell 7.x on Windows

Windows is the platform where most PowerShell users begin their journey, and there are several methods to install PowerShell 7.x. We will cover the most common and recommended approaches.

2.2.1 Installing via the MSI Package

The MSI installer is the most traditional method and provides a graphical installation experience. This is often preferred in enterprise environments where group policy can deploy MSI packages.

First, navigate to the official PowerShell GitHub releases page at `https://github.com/PowerShell/PowerShell/releases`. Look for the latest stable release (at the time of writing, this is PowerShell 7.5.x). Download the MSI file appropriate for your architecture. For most modern systems, this will be the `PowerShell-7.5.x-win-x64.msi` file.

Once downloaded, run the installer. During the installation, you will be presented with several optional features:

Installation Option	Description	Recommended Setting
Add PowerShell to PATH	Adds the pwsh.exe directory to your system PATH environment variable	Enable this option
Register Windows Event Logging Manifest	Enables PowerShell 7.x to write to Windows Event Log	Enable this option
Enable PowerShell Remoting	Configures WinRM for PowerShell 7.x remoting	Enable if you plan to use remoting
Add "Open here" context menus	Adds right-click context menu entries in Windows Explorer	Optional but convenient
Add "Run with PowerShell 7" context menu for .ps1 files	Allows you to right-click a script and run it with PowerShell 7.x	Enable this option

After installation completes, you can verify the installation by opening a new terminal window and typing:

```
pwsh --version
```

You should see output similar to:

```
PowerShell 7.5.2
```

2.2.2 Installing via Windows Package Manager (winget)

For those who prefer command-line installation, the Windows Package Manager provides a clean, scriptable approach. Open a command prompt or Windows PowerShell 5.1 and run:

```
winget install --id Microsoft.PowerShell --source winget
```

This command downloads and installs the latest stable version of PowerShell 7.x. The `winget` tool handles all the configuration automatically, including adding PowerShell to your PATH.

To install a preview version instead, use:

```
winget install --id Microsoft.PowerShell.Preview --source winget
```

Note: The stable and preview versions can be installed simultaneously. The preview version installs to a separate directory and uses the executable name `pwsh-preview.exe`, so there is no conflict.

2.2.3 Installing via the Microsoft Store

PowerShell 7.x is also available through the Microsoft Store. Simply search for "PowerShell" in the Store application and install it. This method has the advantage of automatic updates, meaning you will always have the latest version without manual intervention. However, Store-installed applications run in a slightly sandboxed environment, which may cause subtle differences in behavior for certain administrative tasks.

2.2.4 Installing via the Dotnet Global Tool

If you already have the .NET SDK installed, you can install PowerShell as a .NET global tool:

```
dotnet tool install --global PowerShell
```

This method is particularly useful for developers who already work within the .NET ecosystem and want a lightweight installation.

2.3 Installing PowerShell 7.x on macOS

PowerShell 7.x runs natively on macOS, and the recommended installation method uses Homebrew, the popular package manager for macOS.

2.3.1 Installing via Homebrew

If you do not already have Homebrew installed, open the Terminal application and run:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Once Homebrew is installed, installing PowerShell is a single command:

```
brew install powershell/tap/powershell
```

After installation, launch PowerShell by typing:

```
pwsh
```

You will see the familiar PowerShell prompt, and you can verify the version with:

```
$PSVersionTable
```

This command outputs a detailed table showing the PowerShell version, the .NET runtime version, the operating system, and other relevant information:

```
Name                           Value
----                           -----
PSVersion                      7.5.2
PSEdition                      Core
GitCommitId                    7.5.2
OS                             Darwin 24.2.0 Darwin Kernel
Version 24.2.0
```