mpak: Distribution and Execution Infrastructure for MCP Server Bundles

Mathew Goldsborough
mat@nimblebrain.ai
www.mpak.dev

Abstract. The Model Context Protocol (MCP) ecosystem has discovery and packaging specifications, but lacks distribution and execution infrastructure. We present mpak: a GitHub Action for building bundles, a registry for artifact resolution, and a CLI for fetching and running servers. mpak distributes inert bundles (no code executes at install time) with built-in configuration schemas and OIDC-based provenance.

1. Introduction

The Model Context Protocol (MCP) [1] defines how AI assistants interact with external tools. An MCP server exposes capabilities that AI systems invoke over stdio or HTTP. The ecosystem has matured rapidly.

- **MCP Specification** [1]: Defines the protocol for tool invocation
- **MCP Registry** [2]: Provides discovery ("what servers exist")
- **MCPB Specification** [3]: Defines the bundle format for packaging servers

However, a gap remains between "a packaging format exists" and "developers can easily build, distribute, and run packaged servers." The MCPB specification defines structure but provides no tooling. Publishers must manually construct bundles, host them on GitHub Releases, and communicate URLs out-of-band.

1.1 The Package Manager Mismatch

Traditional package managers appear to solve distribution, but they are designed for a fundamentally different use case:

Characteristic	**Libraries (npm/pip/uv)**	**MCP Servers**
Integration	Imported into code	Run as subprocess
Dependencies	Shared tree, resolved	Vendored, isolated
Execution	Called as functions	Spawned as process
Platform	Often agnostic	May need native builds
Configuration	Via code or env vars	API keys, settings
Lifecycle	Part of app deploy	Independent service

Table 1: Libraries vs MCP servers: fundamental differences in usage patterns

When a developer runs `npm install some-mcp-server`, they get source code in `node_modules/` that they must figure out how to execute. When they run `pip install some-mcp-server`, they get a package in their Python environment that may conflict with their application's dependencies.

MCP servers are not libraries. They are standalone services. The mental model should be closer to Docker images or binary distributions than to npm packages.

1.2 What MCPB Provides (and Lacks)

The MCPB format addresses the packaging problem well:

- **Vendored dependencies**: All dependencies bundled in `deps/` or `node_modules/`
- **Manifest-driven execution**: `mcp_config` specifies command and arguments
- **Platform tagging**: Bundles can be tagged with OS and architecture
- **User configuration**: `user_config` schema for API keys and settings

But MCPB is a format specification, not an ecosystem. It lacks:

- **Build tooling**: No standard CI/CD integration for producing bundles
- **Distribution infrastructure**: No registry for hosting and resolving bundles
- **Execution tooling**: No CLI for fetching, caching, and running bundles
- **Configuration management**: No mechanism for storing user credentials

1.3 mpak's Contribution

mpak, developed by NimbleBrain [4], provides the missing infrastructure layer:

1) **mcpb-pack** [5] (GitHub Action): Automates bundle creation in CI/CD, including dependency vendoring and multi-platform builds.
2) **mpak.dev** [4] (Registry): Indexes bundle metadata, resolves platform-specific artifacts, and provides download URLs. Complements the MCP Registry (discovery) with distribution.
3) **mpak CLI** [6]: Fetches bundles, manages local cache, handles user configuration, and executes servers with proper environment setup.

Together, these components turn MCPB from a specification into a usable ecosystem.

2. Background

2.1 The MCP Ecosystem

The Model Context Protocol standardizes how AI assistants invoke external tools. An MCP server exposes:

- **Tools**: Functions the AI can call (e.g., "query_database", "send_email")
- **Resources**: Data the AI can read (e.g., file contents, API responses)
- **Prompts**: Templated interactions the AI can use

Servers communicate over stdio (for local execution) or HTTP (for remote deployment). The protocol handles capability negotiation, request/response framing, and error handling.

2.2 The MCP Registry

The official MCP Registry [2] provides discovery. Publishers submit server metadata including name, description, repository URL, and declared capabilities (which tools and resources the server exposes). The registry aggregates this into a searchable catalog.

The registry answers "what servers exist?" but does not answer "how do I install and run them?" A registry entry links to a source repository, not to a downloadable artifact. Installation instructions vary by server: some require `npm install -g`, others need `pip install` into a virtual environment, and some expect users to clone the repository and run build scripts. This heterogeneity creates friction, particularly for users unfamiliar with the server's underlying runtime.

2.3 MCPB Bundle Format

MCPB [3] defines a portable package format:

```
bundle.mcpb (ZIP archive)
├── manifest.json
├── src/
├── deps/
└── node_modules/
```

The manifest is the core of the format. It specifies how to execute the server and what configuration it requires:

```json
{
  "name": "@org/postgres",
  "version": "1.2.0",
  "mcp_config": {
    "command": "python",
    "args": ["-m", "postgres.server"],
    "env": { "PYTHONPATH": "deps/" }
  },
  "user_config": {
    "connection_string": {
      "type": "string",
      "description": "PostgreSQL connection URL",
      "sensitive": true,
      "required": true
    }
  }
}
```

The `mcp_config` section tells the runtime exactly how to spawn the server. The `user_config` section declares what credentials or settings the server needs, with type information and sensitivity flags that allow tooling to handle secrets appropriately.

MCPB solves the “what should a bundle contain?” question. It does not solve “how do I build one?” or “where do I get one?”

2.4 Why Not Use npm/pip/uv?

One might ask: if an MCP server is written in Python, why not publish it to PyPI? These tools are designed for **libraries that integrate into application code**, not **standalone servers that run as subprocesses**.

Isolation. Package managers create shared dependency trees. If `mcp-server-a` requires `requests==2.28` and `mcp-server-b` requires `requests==2.31`, they conflict. Using pip for MCP servers means managing separate virtual environments per server. MCPB bundles vendor dependencies per-server, avoiding conflicts entirely without environment management: `mpak run @org/postgres`.

Execution. `pip install mcp-server` gives you a package. How do you run it? The user must discover the entry point, find the correct interpreter, construct the command, set up environment variables, and handle platform-specific dependencies like native extensions. MCPB bundles include `mcp_config` that specifies exactly how to execute, and can include pre-built platform-specific binaries.

Configuration. npm, pip, and uv have no concept of “this package needs an API key.” MCPB’s `user_config` schema declares required configuration with types and sensitivity flags, enabling tooling to prompt for credentials and store them securely.

Trust. Installing via pip can execute arbitrary code (`setup.py` runs during installation). MCPB bundles are inert ZIP files. No code executes until you explicitly run the server, and even then, the user makes the explicit choice to start a named server.

3. System Design

3.1 Architecture Overview

mpak consists of three components that address different phases of the bundle lifecycle:

BUILD PHASE: mcpb-pack GitHub Action
1) Vendor dependencies (Python: uv pip install; Node: npm install)
2) Create bundle (mcpb pack)
3) Upload to GitHub Releases
4) Announce to registry (OIDC-authenticated)

↓

DISTRIBUTION PHASE: mpak.dev Registry
Index metadata, track multi-platform artifacts, resolve downloads, verify provenance

↓

EXECUTION PHASE: mpak CLI
Search, download, cache, configure, execute

3.2 End-to-End Example

To illustrate the complete workflow, consider a developer publishing an MCP server and a user consuming it:

Publisher (Alice):

```
# Alice creates an MCP server with a pyproject.toml
# She adds a GitHub Actions workflow:
# .github/workflows/release.yml
on:
  release:
    types: [published]
permissions:
  contents: write
  id-token: write
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: NimbleBrainInc/mcpb-pack@v2
```

When Alice creates a GitHub release (e.g., `v1.0.0`), the action automatically builds the bundle, uploads it to the release, and announces it to the registry.

Consumer (Bob):

```
# Bob searches for Alice's server
$ mpak search weather
  @alice/weather v1.0.0  "Weather data via OpenWeatherMap"

# Bob runs it (first run downloads and caches)
$ mpak run @alice/weather
=> Pulling @alice/weather@1.0.0...
=> Missing required config: api_key
? Enter api_key: ********
=> Cached at ~/.mpak/cache/alice-weather/1.0.0/
[Server starts]

# Subsequent runs use cached bundle and stored config
$ mpak run @alice/weather
[Server starts immediately]
```

No manual dependency installation. No virtual environment setup. No figuring out how to execute. Bob runs Alice's server with a single command.

3.3 Build Phase: mcpb-pack

The `mcpb-pack` GitHub Action automates bundle creation:

```
name: Release
on:
  release:
    types: [published]

permissions:
  contents: write
  id-token: write

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: NimbleBrainInc/mcpb-pack@v2
```

The action performs runtime detection, dependency vendoring, bundle creation, release upload, and registry announcement via OIDC.

3.4 Multi-Platform Distribution

MCP servers written in pure Python or JavaScript can run on any platform with the appropriate interpreter. However, servers that depend on native extensions (NumPy, cryptography, database drivers with C bindings) require platform-specific builds. A bundle compiled on Linux x64 with native dependencies will not run on macOS ARM64.

This is analogous to the challenge Docker faced with multi-architecture images. Docker's solution was the **manifest list** (later standardized as **OCI image index** [7]): a single logical image that references multiple platform-specific variants. When a user pulls `postgres:latest`, Docker automatically selects the variant matching their platform.

mpak adopts the same pattern for MCPB bundles.

3.4.1 Index Manifest Format

Each package version can have an **index manifest** that describes all available platform variants:

```json
{
  "index_version": "1",
  "mimeType": "application/vnd.mcpb.index+json",
  "name": "@org/server",
  "version": "1.2.0",
  "bundles": [
    {
      "mimeType": "application/vnd.mcpb.bundle+zip",
      "digest": "sha256:a1b2c3...",
      "size": 19456000,
      "platform": { "os": "linux", "arch": "x64" },
      "urls": ["https://github.com/.../linux-x64.mcpb"]
    },
    {
      "mimeType": "application/vnd.mcpb.bundle+zip",
      "digest": "sha256:d4e5f6...",
      "size": 18892000,
      "platform": { "os": "darwin", "arch": "arm64" },
      "urls": ["https://github.com/.../darwin-arm64.mcpb"]
    }
  ]
}
```

The `mimeType` field distinguishes index manifests (`application/vnd.mcpb.index+json`) from actual bundles (`application/vnd.mcpb.bundle+zip`). The `digest` provides content-addressable integrity verification. Each bundle entry includes its platform tuple and one or more download URLs.

3.4.2 Parallel Build Convergence

Multi-platform builds use GitHub Actions' matrix strategy to run builds in parallel across different runners:

```yaml
jobs:
  build:
    strategy:
      matrix:
        include:
          - os: ubuntu-latest
            arch: x64
          - os: ubuntu-24.04-arm
            arch: arm64
          - os: macos-latest
            arch: arm64
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v4
      - uses: NimbleBrainInc/mcpb-pack@v2
```

Each runner independently builds, uploads, and announces its artifact. The registry's announce endpoint is **idempotent**: if multiple runners attempt to announce the same version, the first creates the version record, and subsequent announcements add their artifacts to it. A 409 Conflict response (artifact already exists) is treated as success.

This design enables embarrassingly parallel multi-platform builds with no coordination required between runners. A three-platform build completes in the time of the slowest single build, not the sum.

Runner	Platform	Artifacts	Time
ubuntu-latest	linux-x64	server-linux-x64.mcpb	2 min
ubuntu-24.04-arm	linux-arm64	server-linux-arm64.mcpb	2 min
macos-latest	darwin-arm64	server-darwin-arm64.mcpb	3 min

Table 2: Parallel matrix builds converge on a single version with multiple artifacts

3.4.3 Platform Resolution

When the CLI requests a bundle, it sends the client's platform (`os` and `arch`) as query parameters. The registry performs resolution:

1) **Exact match**: If an artifact exists for the requested platform (e.g., `darwin-arm64`), return it.
2) **Universal fallback**: If no exact match exists but a universal bundle (`any-any`) is available, return it. Universal bundles contain no native dependencies.

3) **Incompatible**: If neither exact nor universal exists, return an error listing available platforms.

```
$ mpak pull @org/server
=> Fetching @org/server (latest)...
   Platform: darwin-arm64
   Version: 1.2.0
   Artifact: darwin-arm64
   Size: 18.02 MB
=> Downloading...
```

This resolution happens server-side, so the CLI never downloads incompatible bundles. Users on unsupported platforms receive clear guidance about which platforms are available.

3.5 Distribution Phase: Registry

The registry (mpak.dev) complements the MCP Registry:

MCP Registry	**mpak Registry**
Lists servers	Hosts bundle metadata
Links to source repos	Links to download URLs
Discovery	Distribution
Source-oriented	Artifact-oriented

Table 3: Complementary roles of MCP Registry and mpak registry

The registry indexes packages, versions, artifacts, and provenance. Importantly, mpak does not store artifacts. Bundles remain on GitHub Releases. The registry indexes metadata and redirects downloads to the source. Each version can have multiple artifacts (one per supported platform), and the registry tracks them all under a unified version identifier.

3.6 Execution Phase: CLI

The `mpak` CLI provides the user-facing interface:

```
$ mpak search postgres
Found 3 bundle(s):
  @nimblebraininc/postgres v1.2.0
  @modelcontextprotocol/postgres v0.9.1

$ mpak run @nimblebraininc/postgres
=> Pulling @nimblebraininc/postgres@1.2.0...
=> Cached at ~/.mpak/cache/nimblebraininc-postgres/
[Server starts, ready for MCP connections]
```

The `run` command handles resolve, download, extract, configure, and execute.

3.7 Configuration Management

MCP servers often require credentials. MCPB's `user_config` schema declares these with types, descriptions, and sensitivity flags. The CLI provides commands for managing server configuration:

```
$ mpak config set @scope/server api_key=sk-xxx
$ mpak config get @scope/server
  api_key: sk-x***
$ mpak run @scope/server  # config used automatically
```

Configuration is stored in `~/.mpak/config.json`, scoped by package name. Values marked as `sensitive` in the manifest are stored in plaintext (the file should be user-readable only) but masked when displayed. At execution time, the CLI resolves each required configuration key using a priority chain: process environment variables take precedence, then stored configuration, then manifest defaults. Environment variables are matched by key name directly (e.g., `api_key` in the manifest is resolved from `$api_key`). If a required key cannot be resolved, the CLI prompts interactively before starting the server.

3.8 Version Management

By default, `mpak run @scope/server` resolves to the latest version. Users can pin to specific versions:

```
$ mpak run @scope/server@1.2.0      # exact version
$ mpak run @scope/server@1.2        # latest patch in 1.2.x
$ mpak list                        # show cached bundles
```

The registry tracks all published versions, and the CLI caches downloaded bundles indefinitely. Cached bundles are never automatically updated; users control upgrades by specifying versions explicitly or running `mpak pull @scope/server` to fetch the latest.

Immutability. Once a version is announced, it cannot be replaced or deleted by the publisher. This prevents "rug pull" attacks where a trusted version is silently replaced with malicious code. Version removal requires registry administrator intervention and is reserved for legal or security incidents.

4. OIDC-Based Provenance

4.1 The Credential Problem

Traditional package registries require publishers to authenticate with long-lived credentials: API tokens stored in CI secrets, environment variables, or developer machines. This creates multiple attack vectors.

Credential theft has affected major registries. Attackers who obtain npm tokens can publish malicious versions of popular packages, as seen in the `event-stream` incident where a compromised maintainer token led to cryptocurrency-stealing code reaching millions of downloads. Similar attacks have targeted PyPI and RubyGems. Even without external attackers, credentials stored in CI systems can leak through log exposure, misconfigured secrets, or insider access.

Beyond security, credentials impose operational burden: tokens must be provisioned per-repository, rotated periodically, and revoked when team members leave. For an ecosystem expecting hundreds of MCP server publishers, this overhead is significant.

4.2 Workload Identity

GitHub Actions provides OIDC tokens [8] that cryptographically assert workflow identity. A token contains claims including repository, owner, commit SHA, workflow, and ref. The token is signed by GitHub and verifiable against GitHub's public keys.

4.3 Announce Protocol

When mcpb-pack announces a bundle:

1) Request OIDC token with audience `https://www.mpak.dev`
2) POST to `/v1/bundles/announce` with token and metadata
3) Registry validates signature against GitHub's JWKS
4) Extract claims: repository, commit, workflow
5) Verify namespace: `@org/name` must come from org's repository
6) Record provenance binding

No credentials are exchanged. Publishers only need:

```
permissions:
  id-token: write
```

4.4 Security Properties

- **No credentials to steal.** There are no API keys or tokens. An attacker cannot publish without controlling the source repository.
- **Provenance is cryptographic.** The binding between bundle and source is signed by GitHub.

- **Namespace ownership.** Packages scoped to `@org/*` can only be published from workflows running in that GitHub organization.
- **Immutable versions.** Once announced, a version cannot be replaced.
- **Namespace governance.** Package scope ownership is delegated to the upstream OIDC provider. Whoever controls a GitHub organization controls the corresponding `@org/*` namespace. Disputes over namespace ownership are resolved through GitHub's existing organization management, not through the mpak registry.

4.5 Threat Model

mpak's security model addresses specific threats while explicitly excluding others:

In Scope:

Threat	**Mitigation**
Credential theft	No credentials exist; OIDC tokens are ephemeral and scoped
Namespace hijacking	Package scopes are bound to GitHub organization ownership
Version tampering	Immutable versions; registry rejects re-announcement
Man-in-the-middle	HTTPS transport; digest verification on download
Build provenance forgery	OIDC tokens are cryptographically signed by GitHub

Out of Scope:

Threat	**Rationale**
Compromised publisher	If an attacker controls the source repository, they can publish legitimately-signed malicious bundles. This is inherent to any system where publishers have autonomy. Mitigation requires external code review and security scanning (see §9).
Runtime sandbox escape	mpak provides process-level isolation only. Malicious servers have full user permissions. Users requiring stronger isolation should use containerized deployment.
Registry compromise	If the registry database is compromised, attackers could redirect downloads to malicious URLs. Mitigation: digest verification ensures content integrity even if URLs are tampered.
Denial of service	Registry availability is a single point of failure. Mitigation: cached bundles continue to work offline.

Trust Boundaries. Users implicitly trust: (1) GitHub's OIDC infrastructure, (2) the mpak registry operator, (3) bundle publishers within their chosen scopes. The system minimizes trust surface but cannot eliminate it.

5. Comparison with Alternatives

5.1 vs. npm/pip/uv

Aspect	**npm/pip/uv**	**mpak**
Distribution unit	Source package	Portable bundle
Dependencies	Resolved at install	Pre-vendored
Execution	User figures it out	`mpak run`
Isolation	Shared environment	Per-bundle
Configuration	None	`user_config`
Install-time code	Yes	No (inert)

Table 6: Comparison of traditional package managers vs mpak

Traditional package managers remain appropriate when MCP servers are tightly coupled to application code, when the development team already manages virtual environments, or when the server has no external dependencies that might conflict. mpak is preferable when servers are deployed independently, when multiple servers with conflicting dependencies must coexist, or when non-developers need to run servers without understanding the underlying runtime.

Startup Performance. The practical difference is most apparent in cold-start scenarios. Traditional package managers must resolve dependencies, download packages, and potentially compile native extensions before execution begins. In testing, `pip install` followed by server startup frequently exceeded 60 seconds for servers with complex dependency trees. npm-based servers showed similar latency when `node_modules/` was not pre-populated.

MCPB bundles eliminate this variability. With dependencies pre-vendored, startup time is bounded by network transfer speed (for uncached bundles) or disk I/O (for cached bundles). A typical 20MB bundle downloads in under 2 seconds on broadband connections; subsequent runs from cache start in milliseconds.

Scenario	**npm/pip/uv**	**mpak**
Cold start (no cache)	30–90 seconds	2–5 seconds
Warm start (cached)	5–15 seconds	<100 ms

Table 7: Representative startup times. npm/pip times include dependency resolution and installation; mpak times include download (cold) or cache lookup (warm).

5.2 vs. Docker

Docker provides execution isolation but is heavyweight:

Aspect	**Docker**	**mpak**
Isolation	Full container	Process-level
Overhead	Container runtime	Direct execution
Image size	Hundreds of MB	Megabytes
Startup time	Seconds	Milliseconds
Runtime req.	Docker daemon	None
Multi-platform	OCI image index	MCPB index manifest

Table 8: Comparison of Docker containers vs mpak bundles

Docker is preferable when servers require system-level isolation, when deploying to container orchestration platforms, or when the runtime environment itself must be controlled. mpak is preferable for local development, for environments where Docker is unavailable or impractical, and when startup latency matters. mpak bundles occupy a middle ground: lighter than containers, more portable than source.

To quantify the size difference, we measured five Python-based MCP servers published as both MCPB bundles and Docker images:

Server	**MCPB Bundle**	**Docker Image**	**Ratio**
echo	19.5 MB	88.6 MB	4.5×
abstract	20.2 MB	88.6 MB	4.4×
ipinfo	20.2 MB	85.7 MB	4.2×
pdfco	20.2 MB	88.7 MB	4.4×
finnhub	19.5 MB	85.9 MB	4.4×

Table 9: Size comparison: MCPB bundles vs Docker images for Python MCP servers

Note: Docker images include a complete OS layer and language runtime, while MCPB bundles rely on the host's existing runtime. This comparison reflects total artifact size, not equivalent isolation guarantees.

On average, MCPB bundles are 4.4× smaller than equivalent Docker images. This difference stems from Docker's inclusion of a full OS layer (Alpine Linux base image, system libraries, Python runtime) while MCPB bundles contain only application code and vendored dependencies, relying on the host's existing runtime.

For Kubernetes deployments, MCPB bundles can run inside lightweight runtime containers. NimbleTools [9], an open-source MCP runtime, provides base images that include only the language runtime:

Runtime Image	**Size**
mcpb-python	41 MB
mcpb-node	54 MB
mcpb-binary	32 MB

Table 10: NimbleTools runtime images for Kubernetes deployment

This separation of runtime from application yields significant savings at scale. Deploying five Python MCP servers traditionally requires five Docker images totaling 445 MB. With NimbleTools, the same deployment uses one shared runtime image (41 MB) plus five bundles (100 MB), totaling 141 MB, a 3.2× reduction.

5.3 vs. MCP Registry Alone

The MCP Registry answers "what servers exist?" but leaves installation to the user. A registry entry might link to a GitHub repository with a README explaining how to clone, install dependencies, and run the server. This works for developers comfortable with the underlying runtime, but creates friction for end users.

mpak complements the MCP Registry by providing the distribution layer. A future integration could allow the MCP Registry to link directly to mpak bundles, giving users a one-command installation path while preserving the registry's role as the canonical discovery mechanism.

6. Implementation

6.1 mcpb-pack Action

The `mcpb-pack` action is implemented as a composite GitHub Action. Composite actions were chosen over JavaScript or Docker actions because they run directly in the workflow's environment, avoiding container overhead and simplifying access to the repository's files.

The action auto-detects the runtime by examining `package.json` (Node) or `pyproject.toml` (Python), then invokes the appropriate dependency installer. For Python, it uses `uv` for fast, reproducible installs into a `deps/` directory. For Node, it runs `npm install --production` to populate `node_modules/`. After vendoring, the action invokes `mcpb pack` to create the bundle, uploads it as a release asset via the GitHub API, and announces to the registry using the workflow's OIDC token.

6.2 Registry Service

The registry is a Fastify server backed by PostgreSQL. Fastify was selected for its low overhead and schema-based validation. The database stores packages, versions, and artifacts as normalized tables with foreign key relationships.

OIDC verification uses the `jose` library [10] to validate JWT signatures against GitHub's published JWKS endpoint. The registry caches the JWKS with a short TTL to handle key rotation. Full-text search uses PostgreSQL's native `tsvector` indexing, avoiding external search infrastructure while providing adequate performance for the expected catalog size.

6.3 CLI

The CLI is a TypeScript application compiled to a single JavaScript file with minimal dependencies: only Node.js and `commander` for argument parsing. This minimizes installation friction and avoids dependency conflicts on user machines.

The CLI stores bundles in `~/.mpak/cache/` with a directory per package-version. Configuration is stored in `~/.mpak/config.json`, with sensitive values kept in plaintext but masked on display. The execution path spawns the server as a child process with `stdio: 'inherit'`, allowing the server's stdout/stderr to flow directly to the terminal.

7. Related Work

7.1 Package Registries

npm [11] pioneered the JavaScript ecosystem but uses credential-based publishing. **PyPI** introduced Trusted Publishers [12] for OIDC-based publishing, which influenced mpak's design. **crates.io** serves Rust with strong security practices.

7.2 Supply Chain Security

SLSA [13] (Supply-chain Levels for Software Artifacts) defines a framework for supply chain integrity. Level 1 requires documented build processes; Level 2 requires hosted builds with authenticated provenance; Level 3 requires non-forgeable provenance from a hardened build platform. mpak achieves Level 3: builds run on GitHub-hosted runners (hardened platform), provenance is cryptographically signed by GitHub's OIDC infrastructure (non-forgeable), and the registry verifies this signature before accepting announcements.

Sigstore [14] provides keyless signing using similar OIDC principles. Where Sigstore focuses on signing arbitrary artifacts with transparency logs, mpak's approach is narrower: it uses GitHub's OIDC tokens specifically for registry authentication, binding package namespaces to repository ownership without requiring publishers to understand signing infrastructure.

8. Limitations

mpak makes deliberate tradeoffs that may not suit all use cases:

Process-level isolation only. Unlike Docker, mpak provides no filesystem or network isolation. A malicious server has full access to the user's system. This is acceptable for trusted publishers but insufficient for running untrusted code. Organizations with strict security requirements should prefer containerized deployment.

GitHub dependency. The OIDC-based provenance model requires GitHub Actions. Publishers using GitLab, Bitbucket, or self-hosted CI cannot currently publish to the registry. Future work may extend OIDC support to additional identity providers.

Runtime requirements. MCPB bundles are not fully self-contained. Python bundles require a compatible Python interpreter on the host; Node bundles require Node.js. Unlike Docker images, which include the complete runtime, mpak assumes the execution environment provides the language runtime. This is typically acceptable for developer machines but complicates deployment to minimal environments.

No transitive dependency resolution. Each bundle vendors its own dependencies independently. If a user runs five servers that all depend on `requests`, they have five copies. This trades disk space for isolation, but may be wasteful for resource-constrained environments.

9. Future Work

- **Registry Federation.** Organizations may want private registries. Future work includes federation protocols.
- **Bundle Signing.** Currently, provenance is established at announce time. Future work could sign bundles for offline verification.
- **Dependency Scanning.** Future work could scan vendored dependencies against vulnerability databases.
- **Security Scanning.** The `mcpb-pack` action could integrate static analysis tools (Semgrep, Bandit, npm audit) to block publication when critical vulnerabilities are detected. The registry could perform post-hoc scanning of announced bundles and flag or quarantine packages with known CVEs.
- **MCP Registry Integration.** The MCP Registry could link to mpak download URLs for servers that have bundles.
- **WebAssembly Bundles.** WASM could enable truly cross-platform bundles without runtime dependencies, eliminating the need for host Python or Node.js interpreters.
- **Offline and Air-gapped Deployment.** Users can already sideload bundles by manually placing `.mcpb` files in the cache directory. Future work includes a `--registry` flag to redirect resolution to internal mirrors, enabling fully air-gapped deployments with synchronized registry endpoints.

10. Conclusion

The MCP ecosystem has a protocol, a discovery registry, and a packaging format. What it lacked was the tooling to build, distribute, and execute bundles. Traditional package managers are designed for libraries, not standalone servers.

mpak fills this gap with three components:

1) **mcpb-pack**: Turns "there's a packaging format" into "here's how to build packages in CI"
2) **mpak.dev**: Turns "host bundles somewhere" into "query for platform-appropriate artifacts"
3) **mpak CLI**: Turns "download and figure out execution" into `mpak run @scope/server`

The system uses OIDC attestation to provide strong provenance without credential management. It complements rather than replaces the MCP Registry, providing the distribution and execution layer that the ecosystem needs.

mpak is available at https://www.mpak.dev. The CLI is `@nimblebraininc/mpak` on npm. The build action is `NimbleBrainInc/mcpb-pack`.

References

[1] Anthropic, “Model Context Protocol Specification”, https://modelcontextprotocol.io, 2024.

[2] Model Context Protocol, “MCP Registry”, https://github.com/modelcontextprotocol/registry, 2025.

[3] Model Context Protocol, “MCPB: MCP Bundle Format Specification”, https://github.com/modelcontextprotocol/mcpb, 2025.

[4] NimbleBrain Inc., “mpak: Distribution and Execution Infrastructure for MCP Server Bundles”, https://www.mpak.dev, 2025.

[5] NimbleBrain Inc., “mcpb-pack GitHub Action”, https://github.com/NimbleBrainInc/mcpb-pack, 2025.

[6] NimbleBrain Inc., “mpak CLI”, https://github.com/NimbleBrainInc/mpak-cli, 2025.

[7] Open Container Initiative, “OCI Image Index Specification”, https://github.com/opencontainers/image-spec/blob/main/image-index.md, 2017.

[8] GitHub, “About security hardening with OpenID Connect”, https://docs.github.com/en/actions/deployment/security-hardening-your-deployments/about-security-hardening-with-openid-connect, 2021.

[9] NimbleBrain Inc., “NimbleTools: Open-source MCP Runtime for Kubernetes”, https://github.com/NimbleBrainInc/nimbletools-core, 2025.

[10] panva, “jose: JavaScript module for JSON Object Signing and Encryption”, https://github.com/panva/jose, 2018–2025.

[11] npm, Inc., “npm Registry”, https://www.npmjs.com, 2010.

[12] Python Software Foundation, “Trusted Publishers”, https://docs.pypi.org/trusted-publishers/, 2023.

[13] OpenSSF, “SLSA: Supply-chain Levels for Software Artifacts”, https://slsa.dev, 2021.

[14] Sigstore, “Sigstore: A new standard for signing, verifying and protecting software”, https://www.sigstore.dev, 2021.