Interposing the Syscall Boundary: Transparent Python Execution in SigmaOS

by

Ivy Wu

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2025

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

© 2025 Ivy Wu. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Ivy Wu
Department of Electrical Engineering and Computer Science
May 9, 2025

Certified by: Ariel Szekely
Ph.D. Student of Computer Science at MIT
Thesis Supervisor

Certified by: M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair
Master of Engineering Thesis Committee

Interposing the Syscall Boundary: Transparent Python Execution in SigmaOS

by

Ivy Wu

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

σOS aims to provide both serverless and stateful support to cloud applications while maintaining strong isolation, security, and efficient startup times and scheduling among multiple users. While σOS and its container startup times have been successfully benchmarked for tasks written, compiled, and statically linked in Golang and Rust, it currently lacks support for other languages, including interpreted ones like Python. To bridge this gap, this paper presents the first integration of an interpreted language into σOS, enabling native Python support without compromising the system's core principles. Our design, σPy, achieves this through three key ideas: (1) system call interposition via `LD_PRELOAD` to enable just-in-time dependency management, where Python libraries are fetched on-demand from tenant-specified AWS S3 buckets, avoiding overhead during container initialization; (2) a multi-layered mount namespace that spans the local machine, a per-realm Docker container, and a per-`proc` σcontainer, enabling efficient dependency caching at the per-tenant granularity; and (3) a hybrid C++, C, and Python API layer that bridges σOS's Protobuf-based RPC system with Python's dynamic types. Preliminary benchmarks demonstrate that σPy achieves performance comparable to that of compiled languages like Golang when interacting with the σOS API, with only 0.2 - 0.3 additional milliseconds of overhead on all tested API calls, validating the success of Python programs on the σOS architecture.

Thesis supervisor: Ariel Szekely
Title: Ph.D. Student of Computer Science at MIT

Thesis supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to express my gratitude to my thesis supervisor and advisor, Professor M. Frans Kaashoek, for his guidance, support, and encouragement throughout my journey towards a Master's degree. I am incredibly thankful for the invaluable feedback that he provided and for the opportunity to learn and grow under his mentorship while pursuing the research directions that I am most interested in.

I am also especially grateful to Ariel Szekely, my mentor throughout this year of research, for his patience, generosity, and constant willingness to help. His advice and expertise with respect to the σOS infrastructure helped me navigate and overcome the technical challenges I faced, and this work would not have been possible without his support.

Thank you both for making this experience such a meaningful part of my academic journey, reinforcing my passion for systems work, and inspiring me to pursue future endeavors in this field throughout the rest of my career.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has become an essential infrastructure for modern applications, offering scalable resources, rapid deployment, and cost efficiency. Among the diverse paradigms, serverless computing platforms like AWS Lambda provide a compelling solution for stateless, ephemeral tasks, while containerized systems such as Docker and Kubernetes enable support for stateful, long-running workloads. Despite these individual strengths, multi-faceted applications that run both types of tasks often face challenges in balancing functionality, security, and performance, leading to compromises in their design.

Take, for example, the startup of new containers in orchestration systems like Kubernetes. These containers are built from large, complex images, which, if not already cached on the machine, require much time to download from a remote server. Once the image has been pulled, the container must be initialized, often requiring setting up the runtime environment, loading relevant data, or even starting long-running background services, all of which are time-consuming tasks that greatly increase the cold start latency. The tradeoff between containerized services and serverless computing lies in the fact that while container designs offer more control and flexibility for complex, stateful workloads, they can introduce higher cold start latencies due to infrastructure and image initialization, resulting in inefficiency and unsuitability for short-lived, stateless functions.

To address this, σOS introduces a unified platform that integrates serverless and stateful capabilities into a single framework. Central to σOS is the concept of lightweight, isolated `proc`s, which ensure rapid initialization, strong isolation, and efficient communication between processes. While σOS has demonstrated its efficacy with compiled languages like Go and Rust, extending its support to interpreted languages such as Python introduces unique challenges. Python's dynamic nature, reliance on interpreters, and need for runtime library management require careful adaptation to the existing `proc` infrastructure without compromising the system's speed, security, and simplicity.

This thesis aims to integrate Python support into σOS, enabling users to leverage Python's rich ecosystem of libraries and frameworks while benefiting from the platform's inherent strengths. The key goals of this work are as follows:

1. **Seamless Integration of the Python Interpreter:** Modifying the σOS build process and container startup to support Python execution without requiring specialized containers or changes to the Python interpreter itself.

2. **Dynamic Library Management:** Developing mechanisms to fetch and manage Python libraries on demand, optimizing storage and performance, in contrast to conventional containerized approaches that require all dependencies to be bundled into the image at startup.

3. **Interfacing with the σOS API:** Creating a C++ and Python-backed library for accessing σOS features such as `proc` control and inter-`proc` communication.

4. **Benchmarking Python Program Performance:** Measuring and analyzing the performance of Python programs running on σOS, including startup time, runtime efficiency, and resource utilization, to compare with alternative cloud computing platforms.

To achieve these goals, this work introduces a new way of running `proc`s into σOS to accommodate Python's dynamic execution model while preserving the existing σOS lightweight isolation model. Unlike container-based solutions that require preloading all dependencies, inflating startup times and storage costs, the σPy framework enables developers to run Python programs that fetch libraries on-demand via standard, unmodified `import` statements via system call interposition, decoupling library management from container initialization. σPy further leverages existing σOS architecture to achieve multi-tenant dependency isolation, creating overlay mounts in σOS's per-tenant realms to store tenant-specific library caches. Finally, σPy offers Python access to σOS platform-specific features, such as `proc` coordination and realm storage, with a C++/Python hybrid library for σOS API access that reconciles Python's types with the σOS RPC system designed in Go. The result is a system where Python programs interact natively with σOS features while behaving exactly as they would outside of σOS, requiring no invasive modifications to the Python interpreter while simultaneously offering the full suite of security, isolation, and performance benefits of σOS.

Our evaluation of 16 critical σOS API calls (12 handling local storage, and 4 interacting with remote AWS S3 storage) demonstrates Python's viability in σOS's isolation model. Python matches Golang in all of the tested functions, maintaining performance within 0.3 milliseconds of the Golang latency through an optimized C++ binding layer for efficient RPC formatting, delivery, and handling. These results validate that Python can meet σOS's performance demands while fully preserving σOS's security guarantees, all without modifications to the Python interpreter itself. This success establishes a template for supporting other interpreted languages in σOS, as the same framework — system-call interception for dynamic imports, hybrid native RPC bindings, and tenant-isolated dependency caching — can generalize to any runtime with similar dynamic linking behavior.

Chapter 2

Background: σOS Design

Modern cloud applications often leverage serverless functions to handle bursts of stateless traffic, while employing containerized services for stateful, long-running tasks. σOS, a multi-tenant cloud operating system, unifies these two needs in a single platform, introducing the concept of `proc`s that are able to start up quickly, maintain state, and communicate with each other [1]. The key abstractions that support these advantages are the σOS API and per-tenant realms, which not only ensure security but also enable inter-`proc` communication and access to filesystem storage, critical for supporting rich user programs and Python integration.

2.1 σOS `proc`s

As a cloud operating system, σOS isolates each user `proc` that it runs, limiting its privileges to prevent any interference with σOS services and other `proc`s, as depicted in Figure 2.1. σOS establishes two layers of isolation with a per-tenant Docker container and a per-`proc` σcontainer. The Docker container enables support for group infrastructure and coordination among `proc`s created by the same tenant, creating a clean, isolated filesystem environment, save for a few mounts from the local machine, and a shared space for establishing communication endpoints. The σcontainer imposes a strict jail on the `proc`, limiting the Linux system calls it can invoke via a seccomp filter and further confining its filesystem access to ensure that most runtime modifications have no effect on the underlying machine.

These σcontainers are light-weight and enable the fast start-up time of σOS in comparison to many other containerized services. Most of the features used by the `proc` require either only a short amount of time to create, such as the seccomp filter and the necessary mounts, or have already been established prior to the `proc`'s initialization, such as the setup done in the per-tenant Docker container. As such, σcontainers ensure privacy and isolation while minimizing the `proc`'s dependency on and usage of the underlying operating system.

σOS `proc`s are able to interact with one another and realm resources via the σOS API. The list of relevant `proc` API calls that σOS provides is shown in Table 2.1.

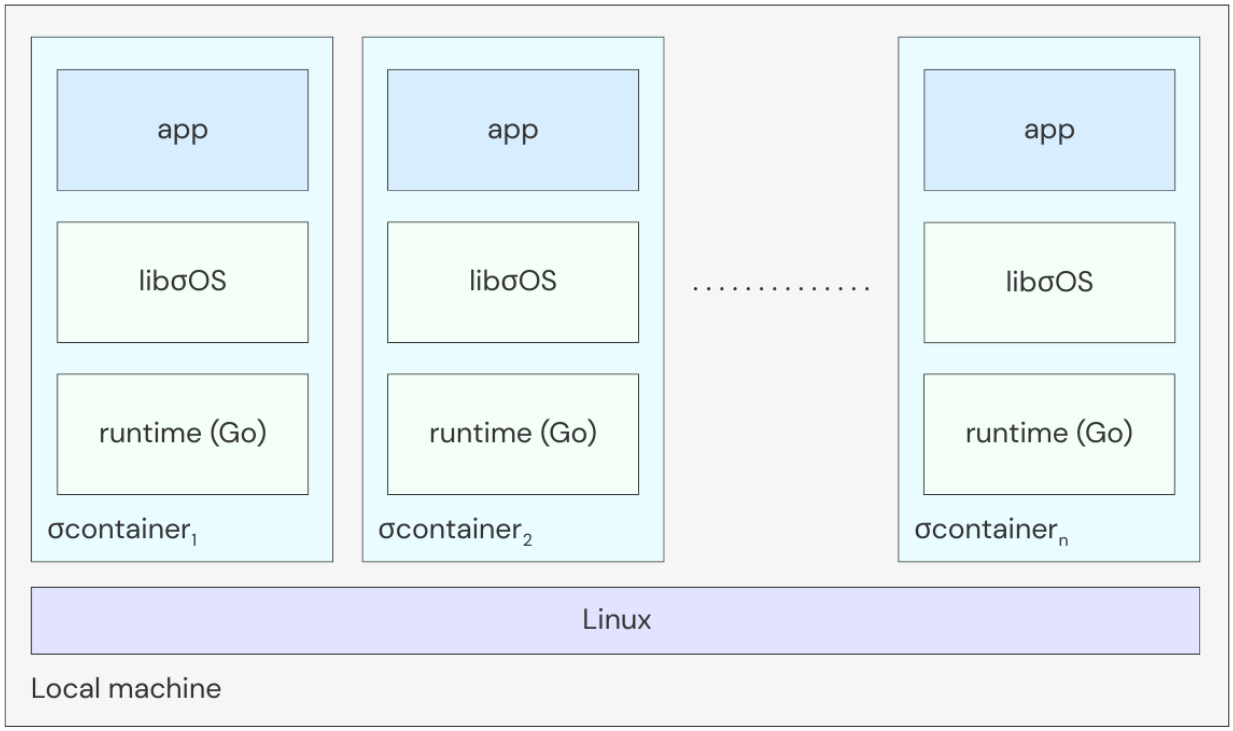

Figure 2.1: `proc` isolation in σOS.

2.2 Inter-`proc` Interactions via Realms

To enable communication between `proc`s invoked by the same tenant, σOS introduces the concept of realms. Each realm allows `proc`s to register endpoints by which they can communicate with each other, called σEPs, and per-`proc` filesystem-like subtrees with unique pathnames that allow `proc`s to recognize and refer to other `proc`s and necessary resources. This allows `proc`s of the same tenant to share information among themselves, helping them coordinate to accomplish tasks.

Each `proc` can create a σEP, which serves as a communication channel accessible by other `proc`s within the same tenant realm. When a σEP is created, it generates a unique token that identifies the endpoint. The creator `proc` can listen on this channel, waiting for and processing any incoming connections. This token can be shared with other `proc`s via the realm's filesystem-like interface, enabling secure and controlled communication and ensuring that `proc`s outside of this realm cannot interfere with the creator `proc`'s activities. Other `proc`s of the same realm can connect to the σEP via the shared token, establishing a messaging channel with the creator and allowing for inter-`proc` coordination and data exchange within the tenant's execution environment.

The per-`proc` filesystem subtrees converge under the same root, a name server called `named`. The same root also holds σOS's proxy services, including ones for `Mongo`, `Db`, `Ux`, and `S3`. These services enable both local and remote storage and retrieval, such as the capability to connect to the tenant's Amazon S3 buckets through the S3 proxy. The data that the tenant would like to use, specific versions of libraries that they depend upon, and backups

Methods	Description
`Spawn(descriptor)`	Queue `proc`, return `pid`
`Evict(pid)`	Notifies a `proc` that it will be evicted
`WaitStart(pid)`	Wait until `pid` starts
`WaitExit(pid)`	Wait until `pid` exits
`Started()`	`proc` marks itself as started
`Exited(status)`	`proc` marks itself as exited
`WaitEvict(pid)`	`pid` waits for eviction notice

Table 2.1: σOS `proc` API Calls

of any important results can thus be efficiently managed and fetched from remote to local storage when necessary, achieving a good balance between network communication overhead and local storage efficiency.

The API enabling these `proc` filesystem changes and realm resource accesses are shown in Table 2.2. Each of the calls may resolve to either the remote or local filesystem depending on the given pathname prefixes — for example, to access the S3 buckets, tenants provide paths prefixed with "name/s3", directing the given request to the S3 proxy to be resolved. The methods for creating, establishing a connection to, and finally closing a σEP are included as well, providing a structured and efficient interface for inter-`proc` communication within the same realm.

Methods	Description
`Create(path, p, m)`	Create a file, directory, link, or pipe with the given permissions
`Open(path, m, w)`	Open the given file, directory, link, or pipe, potentially waiting for the path to be created
`Stat(path)`	Fetch information on an object
`CloseFd(fd)`	Close an object `fd`
`Rename(srcpath, dstpath)`	Rename an object
`Remove(path)`	Remove an object
`GetFile(path)`	Return of contents of an object
`PutFile(path)`	Overwrite the contents of an object
`Read(fd, b)`	Read data from `fd`
`Write(fd, b)`	Write data to `fd`
`Seek(fd, o)`	Change offset of an `fd`
`DirWatch(fd)`	Watch for changes in the given directory
`NewSigmaEP()`	Create a σEP
`Accept(Listener)`	Accept a σEP connection
`Dial(SigmaEP)`	Connect to a σEP
`Close(SigmaEP)`	Close a σEP

Table 2.2: σOS Namespace API Calls

Chapter 3

Implementation of Python `procs`

σPy seeks to integrate Python support into σOS while maintaining the security level and isolation principles of the current `proc` design and without having to adapt the original Python interpreter specifically for the σOS architecture. Changing the Python interpreter itself would introduce significant risks, including the potential for introducing bugs and deviations from expected behavior. Additionally, modifying the interpreter would complicate the integration of future Python versions into σOS, requiring reincorporation of σOS-specific changes and creating an ongoing maintenance burden. At the same time, Python `procs` should integrate smoothly into the σOS architecture and be able to utilize and call the σOS API, which includes important functions like spawning and controlling child `procs`, enabling inter-`proc` communication, and handling access to realm files and folders.

Figure 3.1 provides a high-level overview of the σPy framework, illustrating how the system calls made unmodified Python interpreter are isolated, intercepted, and handled by σOS and enabling communication with outside services like remote AWS S3 buckets. These design goals, constraints, and workarounds are explored in depth in the following sections.

3.1 Python Program Workflow

Python is an interpreted language, where code is not directly compiled into machine language but is instead executed line-by-line by the Python interpreter. The interpreter acts as the runtime engine, processing each line of Python code and converting it into instructions that the computer can execute. This dynamic execution provides significant flexibility compared to compiled languages — Python supports dynamic typing, where variable types can change during runtime, and interactive execution, allowing developers to quickly experiment with, test, and modify small snippets of code.

We introduce a sample Python program, `my_file.py`, to demonstrate the Python workflow. The contents of the file are shown in Figure 3.2.

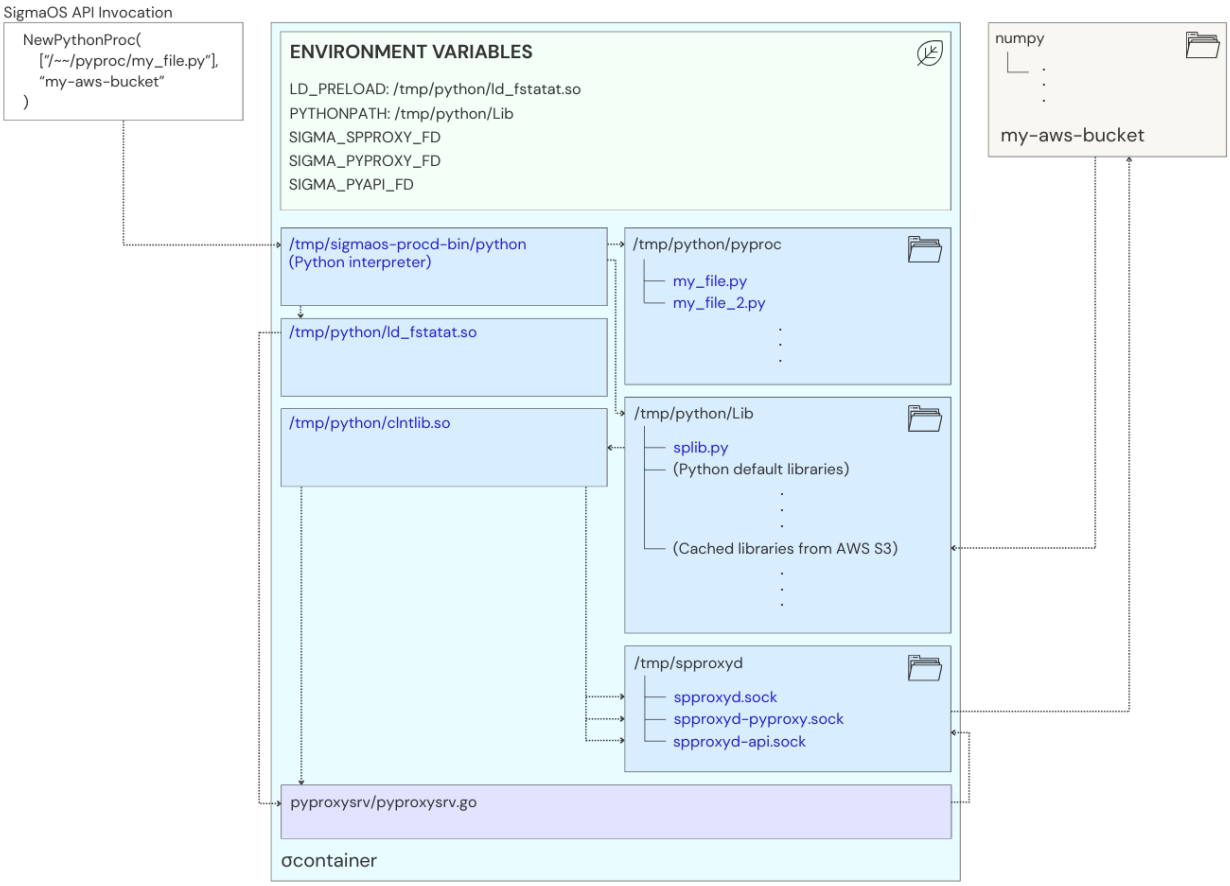

Figure 3.1: High-level overview of σPy.

```
1 import numpy
2
3 print(numpy.array([1, 2, 3]))
```

Figure 3.2: Contents of `my_file.py`

One powerful feature of the Python interpreter is its ability to handle dynamic imports, enabling the program to load external libraries or modules at runtime. For example, `my_file.py` imports `numpy` module at the beginning of the program. When `import numpy` is executed, the interpreter searches for the module through a series of directories, including the current working directory, Python's standard library paths, and locations specified by environment variables like `PYTHONPATH`. This lookup process allows Python programs to remain lightweight, loading libraries only when they are actually needed during execution.

To help with the installation of these libraries, Python comes with `pip`, a built-in package manager that simplifies the maintenance of third-party libraries. Developers can use `pip` to download and install specific versions of third-party libraries such as `numpy` directly from their official sources at the Python Package Index, allowing them to easily access Python's vast ecosystem of external packages and supporting complex projects in machine learning,

web development, and more.

While these features make Python a popular choice for developers, they also present challenges when integrating Python with σOS. It would be impossible for σOS to cache all versions of the hundreds of thousands of Python libraries available, given their sheer volume and constant updates — for example, even without counting patch updates, `numpy` alone has had 16 minor releases [2] [3]. Python's support for dynamic imports introduces further complications, as σOS was designed to support mainly compiled, statically-linked executables for languages like Rust and Go. To handle dynamic imports, σPy must monitor the runtime behavior of Python programs, detect import attempts, and distinguish them from unrelated system calls made by either the Python runtime or other programs. It further must located the imported library and store it locally for future usage, requiring interaction with the σOS infrastructure. This interaction is facilitated by key aspects of the σOS design, such as realm-based storage, allowing Python packages to be persisted in an isolated, per-tenant filesystem.

In running Python via σPy, we aim not only for compatibility but for enhanced functionality through the σOS API. Python `proc`s should be able to access realm resources and coordinate with other `proc`s, necessitating the creation of custom Python libraries to interact with the σOS API. These key goals make designing σPy a challenging engineering exercise, requiring careful design to cohesively integrate Python's unique advantages with the multi-`proc` environment of σOS.

3.2 Running the Python Interpreter

In order to run any Python programs, all users must be able to access the Python interpreter. This required substantial changes to the σOS build process across both the per-realm Docker container and per-`proc` σcontainer. First, σOS must download and compile CPython 3.11 — the only Python version that σOS currently supports — to generate the interpreter file itself and copy it to a location accessible to all users. The interpreter introduces further dependencies on other files and libraries that must be present in the same location; this includes information on the Python build, a file specifying Python PATH, and the Python standard libraries present in the original CPython source code [4]. All of these necessary directories and files must be aggregated into a new Python directory and mounted into the build and eventually `proc` containers for ease of access once a `proc` is run.

Figure 3.3 shows the mounting process from the local machine to the per-realm Docker container (`dcontainer`), previously described in section 2.1. To support Python's dynamic import system, an overlay directory is created from three base directories: `/python/lower`, `/python/upper`, and `/python/work`. CPython 3.11 is downloaded to the local machine and built before being bound as a read-only mount to the `/python/lower` directory. This provides the basic Python interpreter, the Python default libraries, and the custom Python library enabling communication with the σOS API. A dedicated Python directory is created on the local machine under the path `/tmp/{kernel_id}` to bind to and keep track of any changes made in the `/python/upper` directory of the `dcontainer`, ensuring that realm changes are persisted and visible to other `proc`s of the same realm. `/python/work` is a newly created directory inside of the Docker container, serving as scratch space of the overlay directory.

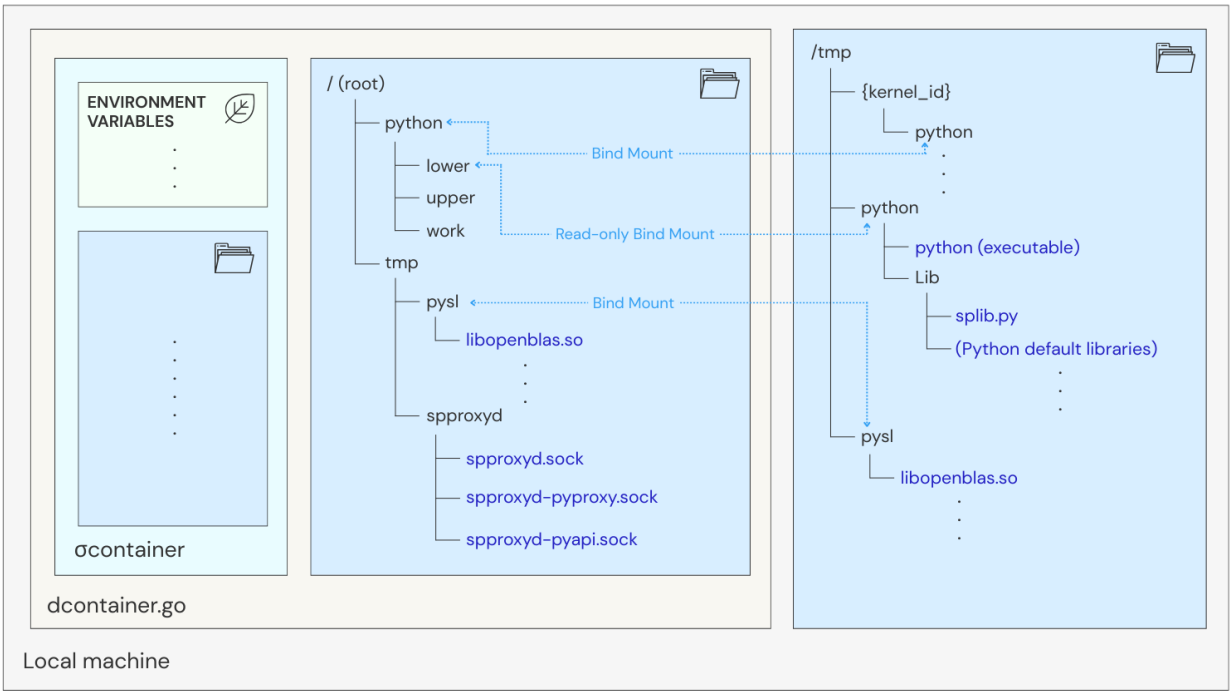

Figure 3.3: The Python-related mounts from the local machine to `dcontainer`.

The final Python-related bind mount inside the `dcontainer`, `/tmp/pysl`, contains any shared Linux libraries that Python depends on, such as the `OpenBlas` library that `NumPy` requires. The only directory created in `dcontainer` without any underlying mount on the local machine, `/tmp/spproxyd`, enables communication with the σOS API, as all writes to the sockets under this directory are processed by the relevant proxy services of the realm.

Figure 3.4 depicts the mounts from the per-realm `dcontainer` to the per-`proc` σcontainer. The `/python/lower`, `/python/upper`, and `/python/work` directories are combined into an overlay mount into the `/tmp/python` directory inside the σcontainer, allowing any `proc`'s changes, such as newly downloaded libraries, to be reflected to all other `proc`s of the same realm. This shared visibility is critical for maintaining both storage and time efficiency: if one `proc` downloads a library from remote storage, subsequent `proc`s can reuse the same it directly, reducing latency and network overhead. The other Docker container directories, `/tmp/pysl` and `/tmp/spproxyd`, are also mounted into σcontainer, allowing the Python `proc` to respectively access its required Linux shared libraries dependencies and make relevant σOS calls.

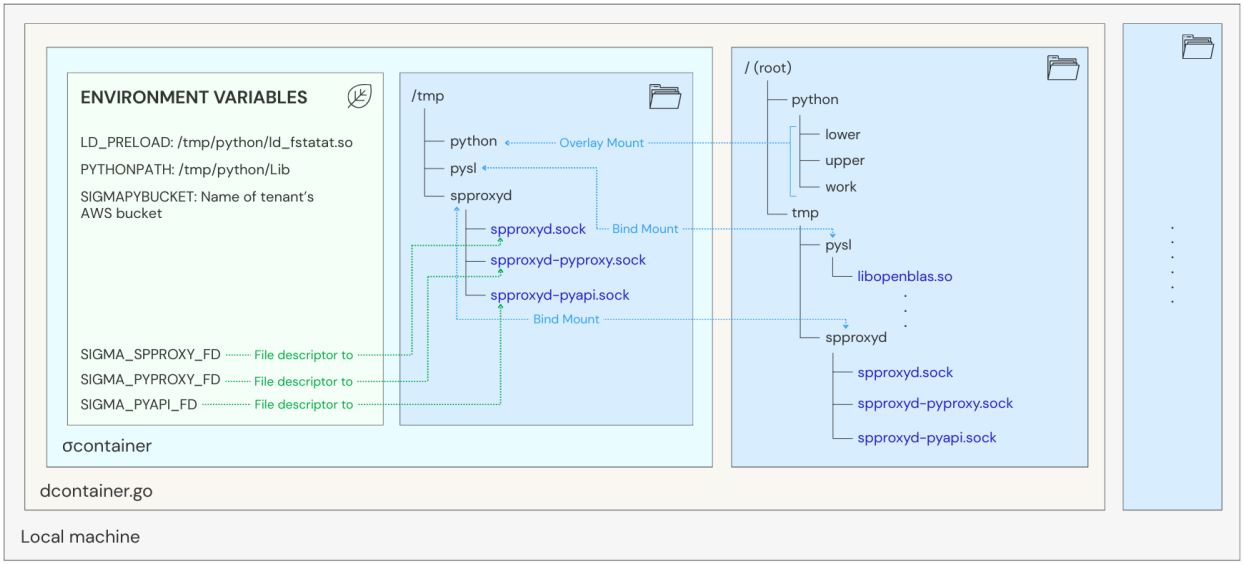

Figure 3.4: The Python-related mounts from the `dcontainer` to the σcontainer.

Supporting the Python interpreter further required an expansion of the system calls allowed by the seccomp filter establishing the `proc` jail. In addition to the 67 system calls currently allowed, Python requires access to the `readlink`, `getcwd`, `gettid`, `stat`, `readv`, and `uname` system calls. All 73 required system calls are shown in Table 3.1.

`ioctl`	`poll`	`lstat`	`clock_gettime`
`membarrier`	`accept4`	`access`	`arch_prctl`
`brk`	`close`	`epoll_create1`	`epoll_ctl`
`epoll_ctl_old`	`epoll_pwait`	`epoll_pwait2`	`execve`
`exit`	`exit_group`	`fcntl`	`fstat`
`fsync`	`futex`	`getdents64`	`getpeername`
`getpid`	`getrandom`	`getrlimit`	`getsockname`
`getsockopt`	`lseek`	`madvise`	`mkdirat`
`mmap`	`mremap`	`mprotect`	`munmap`
`nanosleep`	`newfstatat`	`openat`	`open`
`pipe2`	`pread64`	`prlimit64`	`read`
`readlinkat`	`recvfrom`	`recvmsg`	`restart_syscall`
`rt_sigaction`	`rt_sigprocmask`	`rt_sigreturn`	`sched_getaffinity`
`sched_yield`	`sendto`	`sendmsg`	`setitimer`
`setsockopt`	`set_robust_list`	`set_tid_address`	`sigaltstack`
`sync`	`timer_create`	`timer_delete`	`timer_settime`
`tgkill`	`write`	`writev`	
`readlink`	`getcwd`	`gettid`	`stat`
`readv`	`uname`		

Table 3.1: Allowed System Calls in `proc` Execution Environment

To let users specify which AWS S3 bucket contains the libraries their Python program depends on, σOS adds a new API call: `NewPythonProc(args []string, bucket string)`. The `args` argument specifies the Python file to run and any supported interpreter flags, while `bucket` tells σOS where to look when dynamic imports of non-standard Python libraries occur. σPy directly leverages σOS's built-in S3 proxy service, which already provides secure, tenant-isolated access to cloud storage. Requiring users to supply their own S3 bucket gives them full control over the set and versions of Python packages available to their program, enabling support for customizations that may not be available through the standard repositories. Inside the σcontainer, this API call ultimately runs the command "`python {args}`", with the system dynamically fetching libraries from the specified bucket as needed.

These changes establish the basic functionality of σPy and prepare the system to import and use the default Python libraries. The aggregation of the Python interpreter and its required files, the expansion of the seccomp filter to allow several more system calls, and the introduction of a new API call facilitate the safe invocation of Python programs as σOS `proc`s.

3.3 Runtime Library Imports

As Python is an interpreted language, all imports of required libraries, be it the default Python libraries or custom ones uploaded by the tenant, must be caught and handled by σOS during the lifetime of the program itself. To intercept all imports, σPy sets the `LD_PRELOAD` environment variable to ensure that a shim library is loaded before any other shared object [5]. This C-based shim library, `ld_fstatat.so` overrides system calls related to fetching and opening files and directories, and must verify, for each path that is passed into it, whether or not the path refers to a potentially not-yet-downloaded Python library based on the pathname alone.

Using a heuristic like checking for the appearance of "python" in the path name may not necessarily catch all desired imports, and it may even interfere with fetching the Python interpreter itself or with files used by non-Python `proc`s that happen to have the matching keywords in its filename. To circumvent this issue, σOS sets the `PYTHONPATH` environment variable to the `/tmp/python/Lib` directory, the directory inside of the σcontainer responsible for storing all of the tenant's Python-related programs, libraries, and files. The Python interpreter, upon encountering an `import` statement in a Python file, first searches for libraries under the directory given by this environment variable. The shim library, upon receiving a path name prepended with `/tmp/python/Lib` delivers the request to a σOS Python proxy service referred to as `PyProxy`, which handles the fetching of remote libraries from the tenant's Amazon S3 buckets if necessary. If the specified file or directory is already present under `/tmp/python/Lib`, the `PyProxy` server simply lets the request proceed as normal. Otherwise, it delivers a request to the σOS S3 proxy service to fetch the library to local storage.

The system call interposition of σPy enables seamless, on-demand importing of both default and tenant-supplied Python libraries in a way that is transparent to the Python interpreter. By leveraging a custom preload library and environment variable configuration, σPy efficiently intercepts import-related system calls and defers to a proxy for library retrieval from and communication with AWS S3, ensuring robust and performant execution of Python

programs within the σOS infrastructure while maintaining isolation between `proc`s of different tenants.

3.4 Library Caching

The mounting infrastructure and fetches of stored S3 libraries to the `/tmp/python/Lib` directory, the contents of which are bind mounted from the local machine, enable per-realm per-machine caching of these tenant-supplied Python libraries. If the tenant started multiple `proc`s that all relied on the same library, the S3 fetching process would only occur once, with the remaining `proc`s gaining access to the same library through the system of bind mounts. To ensure that σPy can correct any library content errors, such as crashes whilst in the middle of fetching the content from AWS, it computes and stores a checksum for each library, both on the remote side and on the local side. If the checksum file does not exist locally, or if the contents do not match that of the remote checksum, σPy redownload the entire library. This checksum applies only to libraries fetched from AWS — default Python libraries and our custom σOS API library are marked with an override and do not need to have their checksums computed.

3.5 Relative Imports

Python allows for relative imports, enabling a script to import other files located in the same directory. In σOS, this poses a challenge because the script may not be executed from its original directory, causing its sibling files to no longer appear as local neighbors. To address this, σPy modifies the `PYTHONPATH` during each `NewPythonProc` invocation by appending all parent and ancestor directories up to but not including the `pyproc` root. As the file to run is prefixed with `/∼∼/pyproc`, all of the new additions to `PYTHONPATH` share the same `/∼∼` prefix and are caught by the `LD_PRELOAD` shim. This ensures that related files are recognized as independent libraries, distinct from the ones present in the tenant's AWS bucket as they have no need for local caching, and can be accessed regardless of where the original Python file is actually executed.

3.6 Communication with the σOS API

To take full advantage of all realm resources, Python programs must be able to interact with the σOS API. As such, σPy relies on a newly created Python library file, `splib.py`, and copies it into the default Python library directory for ease of access from all Python programs and files.

σPy's initial implementation of `splib.py` took advantage of the convenience provided by the preload library. Instead of having the Python library communicate directly with each service via their dedicated Unix sockets and RPC handlers, all interactions are funneled through the `PyProxy` service by issuing open system calls on specially formatted pathnames prefixed with "`/∼∼/api`". Figure 3.5 shows the Python code snippet for the initial version of the `Started` API call.

```python
import os

def Started():
    try:
        fd = os.open('/~~/api/Started', os.O_RDWR, 0o666)
    except:
        pass
```

Figure 3.5: `pylib Started` API Call

This approach, however, is unideal as it introduces unnecessary complexity. The σOS architecture already provides a Unix socket under the pathname `/tmp/spproxyd/spproxyd.sock`, which `procs` can use to interact with proxy services and access the σOS API. This socket serves as the entry point for all RPC requests to the API, and Python `procs` can interface with it directly, eliminating the need for an additional layer of proxying for σOS API calls alone.

To support this direct communication, a C++ library is used to construct and format the RPC requests. C++ was chosen specifically to facilitate integration with the existing Protobuf definitions, enabling reuse of the message formats already employed by the σOS RPC infrastructure. As a result, requests issued by Python `procs` are indistinguishable from those issued by native Go `procs` and are handled uniformly.

A C wrapper was implemented around the core functionality to support Python's usage of the C++ library. While Python's native `ctypes` module can integrate smoothly with C libraries, it lacks support for C++ features, such as custom class representations and name mangling during the compilation process [6]. The C wrapper exposes a clean interface for Python to interact with, performing the necessary type conversions between the complex C++ classes generated by the Protobuf definitions to plain C types. This design allows the final shared object generated from the C wrapper, `/tmp/python/clntlib.so`, to be correctly loaded and called from `splib.py` using `ctypes`.

The evolution of σPy's API integration from the initial `/∼∼/api` approach, designed to trigger intervention from the `PyProxy` service, to the direct Unix socket communication now employed demonstrates how σOS's existing RPC infrastructure can be efficiently adapted for Python. Through a carefully designed C++/C binding layer, σPy achieves direct integration with σOS's RPC infrastructure, maintaining Protobuf compatibility for Python while eliminating redundant proxying layers and ensuring that Python, Go, and Rust `procs` share identical RPC formatting and transport semantics.

3.7 σOS Python `proc` Workflow

We modify `my_file.py` to act as a Python `proc`, importing `splib.py` to interact with the σOS API while otherwise maintaining its original functionality. The σOS-compatible version of `my_file.py` is shown in Figure 3.6.

```python
import splib
import numpy

splib.Started()
print(numpy.array([1, 2, 3]))
splib.Exited()
```

Figure 3.6: Modifications made to `my_file.py`

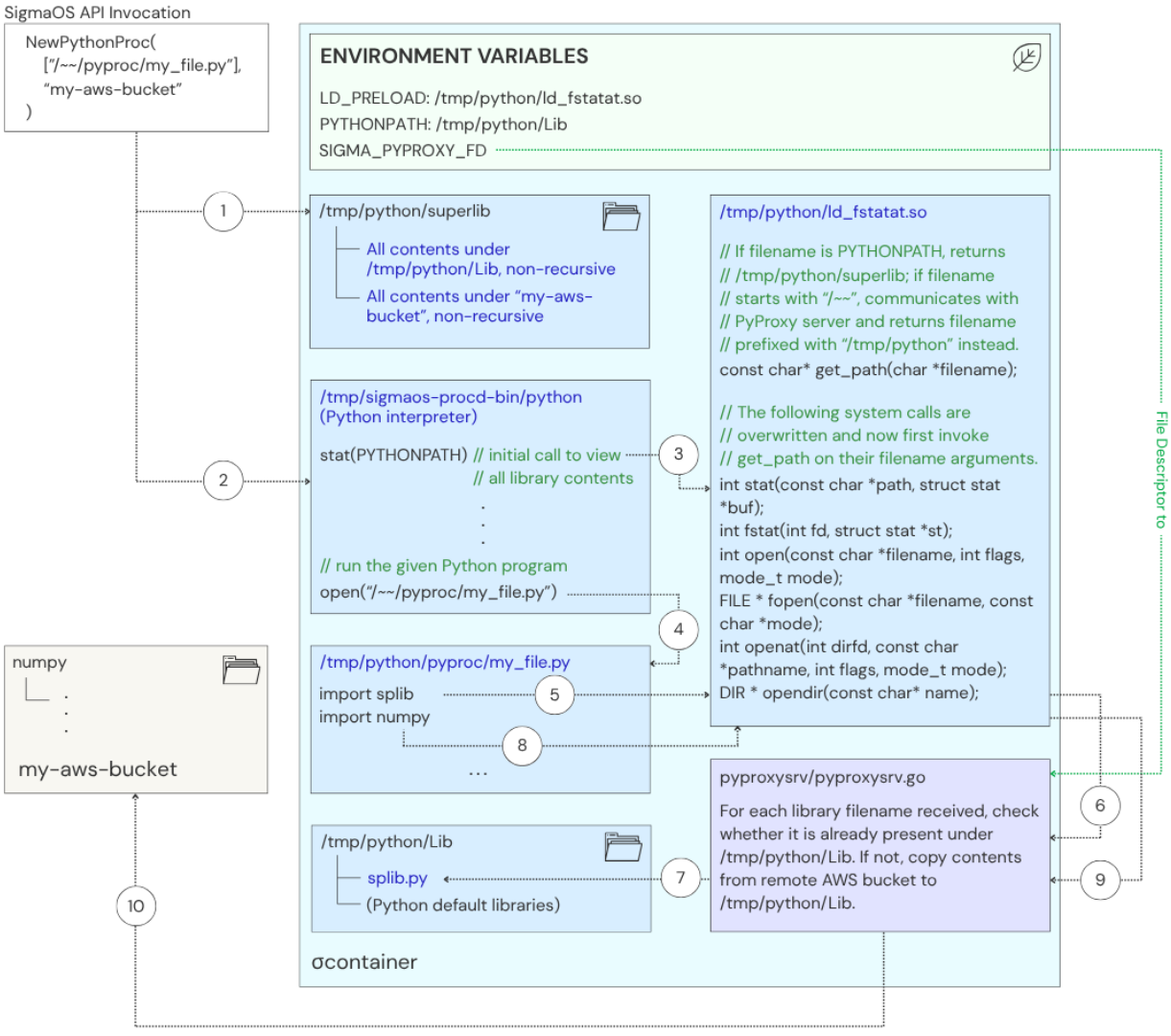

Figure 3.7: `my_file.py` Python `proc` workflow.

The workflow of `my_file.py` under σPy is shown in Figure 3.7. The steps taken by σPy after the creation of a new Python `proc` are as follows:

1. Inside of the σcontainer, σOS creates a directory under the path `/tmp/python/superlib`,

which contains a superset of the names of the direct contents of `/tmp/python/Lib` and the ones of the user-provided AWS S3 bucket. Note that, to reduce startup latencies and memory usage, the true library contents are not copied to this directory; for each library, `superlib` contains just a file of the same name.

2. The Python interpreter is invoked as a `proc` and searches for all available libraries at `PYTHONPATH` (`/tmp/python/Lib`), making system calls like `stat` to the given directory to `mmap` the directory contents for ease of later access.

3. The system call gets intercepted by the shim library, `ld_fstatat.so`. To ensure that all libraries, including the ones uploaded to the tenant's S3 bucket, are seen, `ld_fstatat.so` redirects the system call to open the contents of `/tmp/python/superlib` instead, as `/tmp/python/Lib` contains only the default Python libraries. This is done transparently, with the Python interpreter still believing that it is reading the contents of `/tmp/python/Lib` instead.

4. The Python interpreter, after completing its initial setup, opens and begins to execute the Python program specified by the tenant. Note that σOS currently requires all Python programs to be located in the `sigmaos/pyproc` folder of the local machine. During the build and `proc` initialization process, these Python programs are copied and mounted into the `/tmp/python/pyproc` directory. All Python filename arguments to `NewPythonProc` must be prefixed with "/∼∼" so that it can be caught by `ld_fstatat.so` while remaining distinguishable from dynamic library imports; for example, to access `sigmaos/pyproc/my_file.py`, `NewPythonProc` would be called with the argument `/∼∼/pyproc/my_file.py` instead.

5. `my_file.py` attempts to import `splib`, referring to the file `splib.py` already present under the `/tmp/python/Lib` directory.

6. The system calls used to read the path `/tmp/python/Lib/splib.py` are caught by `ld_fstatat.so`, which examines all pathnames to identify which ones are prefixed by `/tmp/python/Lib`. For these matching pathnames, it writes them to the `PyProxy` Unix socket, whose file descriptor is stored in the `SIGMA_PYPROXY_FD` environment variable, and blocks until it receives a response. This file descriptor points to a Unix socket that the `PyProxy` service reads from and writes a response back to.

7. The `PyProxy` service examines the contents of the local `/tmp/python/Lib` directory and sees that `splib.py` is stored there already, allowing the import command to return and for `my_file.py` to continue executing.

8. `my_file.py` then seeks to import `numpy`, a non-default library located in the tenant's S3 bucket. The Python interpreter first checks to make sure this library exists, which we have ensured is true via the creation of `superlib`. Then, it seeks to actually open and examine the contents of `/tmp/python/Lib/numpy`.

9. Similar to what happened with the `splib` import, the system calls used to read from the `/tmp/python/Lib/numpy` directory are caught by `ld_fsatat.so` and forwarded to the `PyProxy` service.

10. The `PyProxy` service, upon receiving the `/tmp/python/Lib/numpy` pathname, sees that it is not yet present under the `/tmp/python/Lib` directory. As a result, it communicates with the S3 proxy service to fetch the `numpy` directory recursively from AWS, storing it in `/tmp/python/Lib` for the tenant to use. It then writes a response back to the original system call that triggered the `ld_fstatat.so` intervention, allowing it and the remaining commands in `my_file.py` to proceed as normal. Libraries of different tenants do not interfere with each other, as their per-`proc` `/tmp/python/Lib` directories bind to different directories on the underlying host machine.

The above workflow demonstrates how dynamic imports are handled within σPy. Having successfully imported `splib.py`, `my_file.py` is able to interact with the σOS API. Figure 3.8 shows how the `Started` API call proceeds under σPy.

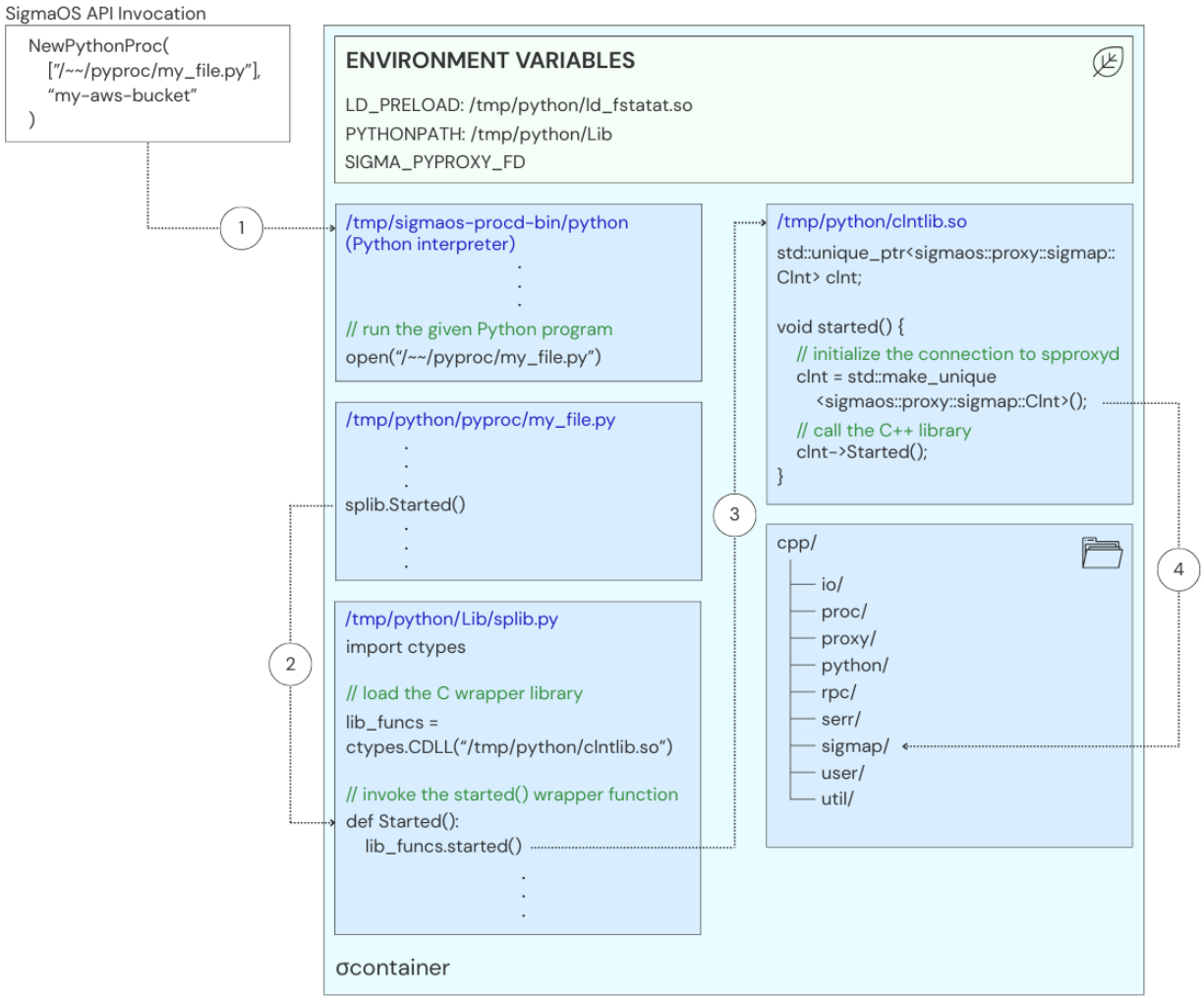

Figure 3.8: Sample Python `proc` interaction with the σOS API.

The `Started` API call is executed as follows:

1. `my_file.py` begins executing after being launched through the `NewPythonProc` API

call, carrying out the previously described setup steps and importing any necessary libraries, including `splib.py`.

2. The `Started` function in `splib.py` acts as a lightweight Python binding, delegating the call to the C wrapper's `started` stub.

3. The C wrapper acts as a bridge between the Python and C++ layers, forwarding the call to the C++ library. In this case, no type conversions are necessary, as this API call has no input parameters or return values.

4. Once invoked, the C++ library establishes a client connection to the Unix domain socket at `/tmp/spproxyd/spproxyd.sock` and issues the corresponding `Started` RPC.

This example demonstrates how a Python program can be securely and efficiently supported within σOS. By leveraging system call interception, on-demand library fetching, and a lightweight API binding layer, σPy maintains the core σOS principles of isolation and security without hindering Python's flexibility as a dynamically interpreted language. This approach avoids making invasive changes to the Python interpreter while ensuring compatibility with existing workflows such as AWS S3-backed remote storage. As a result, Python programs have full access to σOS features, including cross-`proc` communication and realm storage, enabling optimizations like library caching to enhance performance while maintaining the same degree of security present across all σOS `proc`s.

Chapter 4

Evaluation

To evaluate the efficiency of Python support in σOS, we seek to assess whether Python interactions with the σOS API achieve comparable performance to its Golang counterpart. A critical distinction between the two implementations lies in the Python library's reliance on the `SPProxy` socket for communication, an extra hop bypassed by the Go version. This difference, as well as Python's nature as an interpreted language, introduces potential overhead. Our evaluation quantifies the latency impact of these Python-specific overheads, testing to see if the latency of Python requests remains competitive with Go for multiple σOS use cases.

4.1 Experimental Setup

We conducted experiments measuring the performance of 12 σOS API calls, which perform a mixture of local storage operations within the realm namespace and RPCs to σOS kernel services. To reduce variance, each API call was executed 50 times per language. All trials were run on a single machine equipped with Intel Xeon CPUs E5-1410 0 that achieve a maximum frequency of 2.80GHz.

The benchmarks were intentionally lightweight, focusing on minimal data transfers to isolate API functionality. For instance, the `PutFile` and `Write` operations used only the 12-byte string "Hello World!" as the data to be written to the specified files.

One benchmark measured purely on the Python side is the `ClntID` operation. This API call is responsible purely for connecting to the `SPProxy` socket and does not perform any filesystem reads or modifications, serving as a reasonable estimate for the additional overhead introduced by Python's reliance on communication through the `SPProxy` Unix domain socket.

4.2 Microbenchmark Results

We evaluated 12 σOS API calls, timing their access to files in the local realm namespace and RPCs made to other services. The latencies measured were averaged across the 50 runs and are presented in Table 4.1.

While Python exhibits slightly higher latency across all API calls compared to Go, the results demonstrate extremely close performance in many cases. Most operations show modest overhead, with Python typically adding up to 0.3 milliseconds of delay to Go's baseline

API Call	Go Latency (ms)	Python Latency (ms)
`Started()`	1.14	1.16
`Exited()`	0.32	0.64
`Create(Filepath)`	0.64	0.92
`Open(Filepath)`	0.47	0.64
`Write(FileDescriptor)`	0.18	0.49
`Read(FileDescriptor)`	0.10	0.42
`Seek(FileDescriptor)`	0.00	0.20
`CloseFD(FileDescriptor)`	0.11	0.28
`PutFile(Filepath)`	0.35	0.68
`GetFile(Filepath)`	0.35	0.77
`Stat(Filepath)`	0.64	0.92
`Remove(Filepath)`	0.34	0.41

Table 4.1: Latency Comparison of Go versus Python API Calls

latency. This matches our results from benchmarking the `ClntID` API call — across all 50 runs, the average latency of this Python operation was 0.22 milliseconds. Collectively, these measurements quantify Python's overhead to be in the range of 0.2 - 0.3 milliseconds for most σOS API calls as a result of the extra `SPProxy` Unix socket that Python must connect to and interact with.

Chapter 5

Related Work

There are a variety of serverless and stateful computing services, such as AWS Lambda, Docker, and Kubernetes.

Docker and Kubernetes are popular container systems, respectively providing support for single nodes and clusters of containers. Docker provides multiple Python base images that support anywhere between versions 3.9 to 3.14, and users can install any desired libraries via pip inside of the Dockerfile [7]. Kubernetes, on the other hand, takes on the role of container orchestration, with individual containers relying on Docker or other types of images. Kubernetes enables communication between these clusters of containers, called pods, through networking, enabling coordination between user processes [8].

AWS Lambda, which originated as a completely serverless platform, initially attracted customers due to its quick start-up times and high load scalability [9]. It currently supports Python versions 3.8, 3.9, 3.10, and 3.11, and requires uploads in the forms of ZIP files containing both the Python source code to run as well as any necessary dependencies [10]. These dependencies include the external packages and modules referenced by the source code. More recently, AWS Lambda has expanded its Python support by allowing users to deploy Python Lambda functions with container images [11]. In this case, AWS Lambda will search for any imported libraries inside of the container image itself.

While platforms like AWS Lambda and Kubernetes rely on packaging all dependencies ahead of time or building them into container images, σOS must detect and fetch dependencies dynamically at runtime. To accomplish this without modifying the Python interpreter, σOS uses system call interposition to monitor and redirect library access attempts made by Python `procs`. This interposition is enabled by the `LD_PRELOAD` environment variable, which injects the specified shared object before any others are loaded, allowing for manual overrides and redefinitions of Linux system calls like `open`, `read`, etc. This technique allows σOS to scan and verify all pathnames accessed by these system calls to catch any dynamic library import attempts, supporting compatibility with the Python interpreter while preserving the expected behavior of all other system operations.

In contrast to existing computing services, which typically require bundling all dependencies into a container image, significantly increasing startup latency, the σOS Python protocol dynamically retrieves Python modules on demand at runtime, enabling fast initialization while preserving the flexibility of dynamic imports. By running the Python interpreter as a `proc`, σOS eliminates the need for specialized containers tailored to Python

programs, ensuring secure execution with minimal setup. The only additional step required during `proc` initialization is mounting the Python-related directories into the `proc` container, granting seamless access to Python's standard libraries. Moreover, σOS enhances flexibility by supporting the on-demand download of libraries during the program's lifetime with its system call interposition, a process that transparently intercepts Python's dynamic import attempts without disrupting unrelated system calls from Python itself or other programs. This approach amortizes library fetching costs over the program's runtime and downloads libraries only when they are explicitly imported, ensuring both efficiency and convenience for users.

Chapter 6

Conclusion

This work enables Python support in σOS through three key ideas. First, σPy leverages system call interposition via an `LD_PRELOAD` shim library to transparently redirect Python's dynamic imports to the `PyProxy` service, which searches for the library in both local and remote storage, fetching and downloading the library if needed. Second, it takes advantage of σOS's existing isolation structures at both the per-realm and per-`proc` level to design multi-layer mount architectures that enable cross-`proc`, per-realm caching and reuse of already downloaded libraries. Third, σPy implements a C++, C, and Python-based API library that all Python `proc`s may import, enabling Python's access to and usage of σOS's existing API calls, such as those for inter-`proc` communication and realm storage. Together, these innovations allow Python programs to be run as σOS `proc`s without requiring any modifications to the Python interpreter and while preserving σOS's isolation guarantees, with benchmark results confirming that σPy allows Python to achieve parity with Go's speed in its interactions with the σOS API.

6.1 Future Work

While the σPy framework demonstrates Python's viability in σOS, several directions remain open for improvement and expansion:

1. Python libraries often depend on specific versions of a Linux shared library. For example, one issue we experienced when testing out `NumPy v1.25.0` was its dependency on `OpenBLAS v0.3.23`. Furthermore, this library required compilation from source with the `SYMBOLSUFFIX=64_` flag to ensure NumPy could recognize all symbols, which must end with `64_`. σOS's Python support requires users to supply these specific dependencies, as pre-downloading all possible shared library versions during the build process would be inefficient when only select versions are needed. However, the Python built-in package manager, `pip`, resolves these dependencies well. As such, integrating `pip` support into σOS could be a direction worth investigating. As an alternative, σOS could offer users a choice between two dependency schemes: those prioritizing convenience could specify the name and version of the library that they want to use, and σOS could use `pip` to download and install these libraries during the dynamic import process, while performance-focused users could continue with the current scheme.

This flexibility would leverage `pip`'s robust dependency resolution while preserving the current system's advantages in handling custom, user-provided Python libraries.

2. The current caching mechanism of σOS is limited to per-machine, per-realm storage of Python libraries, which provides multiple opportunities for optimization. We can extend caching across multiple machines by taking advantage of σOS's existing file sharing infrastructure, which is already used for sharing `proc` binaries. Additionally, independent tenants may share dependencies on the same library versions. If these libraries are already present on the machine as a result of another tenant's download process, and we can ensure that this library contains no custom modifications — for example, if we knew that it was downloaded through the `pip` scheme mentioned in the previous point — we can change the mounting process to allow for reuse of these libraries as well.

3. The design developed for Python — combining system call interception for dynamic imports, mount-based isolation, and C-compiled shared library bindings — establishes a reusable framework for other interpreted runtimes. Languages like Ruby or R could leverage similar mechanisms, with their just-in-time dependency resolutions being managed by σOS's shim library as well.

The σPy framework proves that interpreted languages can integrate seamlessly with σOS's isolation model while preserving existing developer workflows and requiring no additional changes to the interpreted languages themselves. By solving Python's unique challenges, such as dynamic imports, per-tenant dependency isolation and caching, and native access to the σOS API, σPy establishes a blueprint for supporting other runtimes like Ruby or Node.js. Future work ono cross-machine caching and `pip` integration will further extend σOS's flexibility and ease of use, solidifying σOS as a unified platform for both compiled and interpreted, stateful and serverless workloads.

References

[1] A. Szekely, A. Belay, R. Morris, and M. F. Kaashoek. "Unifying serverless and microservice workloads with SigmaOS". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pp. 385–402. ISBN: 9798400712517. DOI: 10.1145/3694715.3695947. URL: https://doi.org/10.1145/3694715.3695947.

[2] P. Stats. *Analytics for PyPI packages*. https://pypistats.org/.

[3] endoflife.date. *NumPy*. https://endoflife.date/numpy.

[4] P. S. Foundation. *The Python Tutorial*. https://docs.python.org/3/tutorial/index.html.

[5] *ld.so(8) — Linux manual page*. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[6] R. Tibbetts. *The Secret Life of C++: Symbol Mangling*. https://web.mit.edu/tibbetts/Public/inside-c/www/mangling.html.

[7] Docker. *python*. https://hub.docker.com/_/python.

[8] T. K. Authors. *Production-Grade Container Orchestration*. https://kubernetes.io/.

[9] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka. *On-demand Container Loading in AWS Lambda*. https://arxiv.org/pdf/2305.13162.

[10] A. W. Services. *Working with .zip file archives for Python Lambda functions*. https://docs.aws.amazon.com/lambda/latest/dg/python-package.html.

[11] A. W. Services. *Deploy Python Lambda functions with container images*. https://docs.aws.amazon.com/lambda/latest/dg/python-image.html.