Qwen

A Critical Analysis of Python's Suitability for High-Assurance Neurotechnology Systems

The Unmitigated Threat of the Python Software Supply Chain

The assertion that the Python software supply chain represents a significant and unmitigated threat is overwhelmingly supported by extensive evidence of persistent, large-scale, and increasingly sophisticated attacks. While Python's vast and accessible package repository, PyPI, has fueled innovation, it has simultaneously created a fertile ground for adversaries seeking to compromise applications through indirect means [31 37]. For a mission-critical system like Project Chimera, which demands absolute integrity and trustworthiness, relying on this ecosystem is akin to building a foundation upon shifting sand. The threats are not isolated incidents but rather a systemic failure of trust that current mitigation strategies have yet to fully resolve. The scale of the problem is staggering; in 2022 alone, the PyPI team was forced to remove over 12,000 unique projects due to malware, indicating a continuous deluge of malicious activity [39]. This volume suggests that any attempt to manually vet every dependency is untenable and that automated defenses are essential but still lagging behind the attackers' ingenuity. The fundamental issue lies in the open nature of the repository, where almost anyone can upload code without mandatory security review, making "trust" a dangerous assumption for any critical application [34 36].

A primary vector for these compromises is typosquatting, a technique that exploits human error during package installation [34]. Attackers register packages with names that are intentionally misspelled versions of popular libraries, such as 'requsets' instead of 'requests' or 'coloramapkgsw' instead of 'Colorama' [34 37 38]. These malicious packages often contain convincing documentation and links to legitimate resources to gain developer trust, leading unsuspecting users to install them under the mistaken belief they are obtaining a legitimate library update or tool [37]. Once installed, these packages can execute a variety of malicious payloads, including harvesting environment variables containing sensitive API keys and tokens, exfiltrating browser data, stealing cryptocurrency wallets, and installing Remote Access Trojans (RATs) that provide full remote control of the infected machine [31 36]. The 'beautifulsup4' package, mimicking 'beautifulsoup4', was designed to hijack clipboard content to replace cryptocurrency wallet addresses with those controlled by the attacker, demonstrating a highly targeted financial motive [38]. Other variants target Windows machines by installing browser extensions that hijack clipboard content to redirect cryptocurrency transactions [38]. The persistence and evolution of this attack method underscore its effectiveness; researchers have identified dozens of typosquatted packages targeting various ecosystems, including npm JavaScript color conversion libraries like 'colorizr' [32 37].

Beyond simple typosquatting, more complex techniques like dependency confusion represent a critical vulnerability in the Python ecosystem [35]. This attack exploits the default logic of package

managers, which often prioritize public repositories like PyPI over private ones. An attacker can publish a malicious public package on PyPI with the same name as a proprietary internal package used by an organization, provided the public version has a higher version number [35 105]. When developers or automated build processes fetch dependencies, the package manager will inadvertently download and install the malicious public version, leading to a mass infiltration of the organization's systems [35]. The compromised PyTorch-nightly dependency chain in December 2022 is a prime example, where a malicious package mimicked a legitimate PyTorch dependency, exposing confidential data on Linux systems [105]. This attack vector is particularly insidious because it leverages the very tools designed to manage complexity and ensure consistency, turning them into a mechanism for delivering compromise.

Another alarming trend is the use of malicious updates, where attackers first publish a seemingly benign package to gain adoption and trust before later pushing a malicious payload [32]. The 'semantic-types' package, used in the development of Solana cryptocurrency wallets, is a case in point. Initially uploaded as a legitimate utility, it received a malicious update that introduced functionality to steal private keys by monkey-patching key-generation methods at runtime without modifying the source code [32 37]. This technique, known as monkey patching, allows attackers to modify functions in a module after it has been loaded, making detection exceptionally difficult as the original source code appears clean [37]. Similarly, the 'solana-live' package, masquerading as a price-fetching library, specifically targeted Jupyter Notebooks for data exfiltration [32]. These supply chain poisoning campaigns demonstrate a long-term strategy of patience and deception, eroding trust over time rather than through a single, obvious attack.

The sophistication of these attacks continues to evolve, moving beyond simple obfuscation to evade detection by modern security tools. Attackers now employ techniques such as using visually similar Unicode characters to disguise function names, dynamically reconstructing function calls through string operations, and encoding payloads with Base64 or zlib to avoid static signature-based detection [31 56]. A novel and particularly concerning attack vector involves hiding malicious code within machine learning models [37 40]. Researchers discovered that three malicious PyPI packages impersonating Alibaba AI Labs SDKs contained no functional AI code but instead delivered malware embedded within PyTorch model files [32 40]. These models were zipped Pickle files, and the malicious payload was hidden inside them [40]. This exploits the common misconception among developers that ML model files are passive data artifacts rather than executable code, allowing the malware to bypass traditional security controls that lack deep inspection capabilities for ML formats [40]. This attack highlights a new frontier in supply chain compromise, leveraging the growing reliance on AI/ML libraries within the Python ecosystem.

In response to this escalating threat landscape, significant efforts have been made to improve the security of the Python supply chain. The introduction of Sigstore-based digital attestations via PEP 740 represents a major step forward, aiming to provide cryptographically verifiable provenance for packages [45 50]. This system binds a package to its source code repository, commit hash, and CI/CD workflow, allowing downstream consumers to verify that a package was built from trusted sources [46 50]. However, the adoption of these crucial defenses remains nascent and faces significant barriers.

As of one report, only about 5% of the top 360 most-downloaded Python projects had published attested packages, largely because many had not been updated since the feature became available [50]. Furthermore, while PyPI hosts and verifies these attestations, the responsibility for verification falls to the consumer, and installation tools like `pip` do not yet perform this check automatically [50]. This creates a critical gap where downstream users must still implicitly trust PyPI to serve the correct, unaltered attestations, a risk that persists even with cryptographic signatures [50]. The broader challenge is that these new security features are not yet deeply integrated into the core developer workflow, creating a self-reinforcing cycle where low demand delays packaging and limits usage [47]. Until verification becomes a seamless, default part of the installation process, the Python supply chain will remain vulnerable to the very attacks it purports to defend against.

Attack Vector	Description	Example Packages	Impact
Typosquatting	Uploading malicious packages with names similar to popular libraries to exploit human error during installation.	`requsets`, `coloramapkgsw`, `beautifulsup4`, `bitcoinlibdbfix` [34 38 107]	Data exfiltration (API keys, credentials), Remote Access Trojan (RAT) installation, Cryptocurrency theft. [31 36 38]
Dependency Confusion	Publishing malicious public packages with the same name as an organization's private internal packages to trick package managers into fetching them.	Not Applicable (technique)	Mass infiltration of organizational systems, compromise of internal build pipelines. [35 105]
Malicious Updates	Initially publishing a benign package to gain trust and downloads, then pushing a malicious update later to introduce backdoors.	`semantic-types` [32 37]	Exfiltration of sensitive data (e.g., Solana private keys). [32]
Cross-Platform Attacks	Exploiting typosquatting across different package registries (e.g., PyPI and npm) to maximize reach.	`colorizator` (PyPI), `colorizr` (npm) [37]	Widespread compromise of developers working in multiple ecosystems. [37]
Payload Obfuscation	Using Unicode characters, dynamic code reconstruction, and encoding to evade static analysis and signature-based detection.	Multiple packages detected by GuardDog and other tools [38 56]	Bypassing automated security scanning tools, increasing the likelihood of undetected malicious code. [56]

Attack Vector	Description	Example Packages	Impact
ML Model Poisoning	Embedding malicious payloads within serialized machine learning models distributed as PyTorch files.	`aliyun-ai-labs-snippets-sdk` [32] [40]	Delivery of malware disguised as legitimate ML components, evading standard security checks. [40]

Architectural Flaws in Python's Design and Runtime Environment

Beyond the pervasive threat of the software supply chain, Python's fundamental design and runtime architecture present a series of profound and often irreconcilable challenges to the creation of high-assurance systems. Its strengths—developer productivity, dynamic flexibility, and an expansive ecosystem—are achieved at the cost of core security tenets such as memory safety, runtime isolation, and predictable performance. For applications demanding absolute reliability and security, such as nanoswarm platforms, these architectural trade-offs are not merely inconveniences but fatal flaws. The language is explicitly not designed to be a secure sandbox, and its powerful features for introspection and dynamic code manipulation can be weaponized to bypass any attempted security boundaries [84]. These issues are compounded by the widespread use of native C extensions, which reintroduce the very memory-safety vulnerabilities that Python's garbage collector is meant to prevent [78] [80].

One of the most critical architectural deficiencies is Python's handling of memory safety. While pure Python code is generally considered memory-safe due to automatic memory management by the interpreter, this guarantee is entirely contingent on avoiding interactions with unsafe code [76]. The reality is that the scientific and machine learning communities, which are central to neurotechnology, heavily rely on libraries written in C and C++ (e.g., NumPy, TensorFlow, SciPy) to achieve performance [78]. These native extensions operate outside the bounds of Python's memory management, reintroducing the entire spectrum of classic memory-safety vulnerabilities: buffer overflows, use-after-free errors, null pointer dereferences, and double free conditions [76] [80]. These types of bugs are responsible for a staggering percentage of critical software vulnerabilities reported by organizations like Microsoft and Google, and their presence in a system controlling physical matter or neural interfaces would be catastrophic [80]. The U.S. Cybersecurity and Infrastructure Security Agency (CISA) and the NSA jointly recommend transitioning to memory-safe languages precisely to eliminate these classes of vulnerabilities at their source [80] [82]. Python, by its very nature, cannot provide this assurance in a mixed-language environment. While solutions exist to sandbox Python, such as running it within Intel SGX enclaves, these are complex workarounds that add significant performance overhead and implementation burden rather than being a built-in feature of the language itself [77].

Equally problematic is Python's inability to provide a reliable runtime sandbox. Core developers have stated unequivocally that sandboxing arbitrary code within the standard CPython interpreter is considered impossible due to the language's deep dynamism [84]. Features that are central to Python's

power and flexibility—such as `eval()`, `exec()`, `__import__()`, and even the `ctypes` module for calling C functions—are fundamentally incompatible with security isolation [54 55 84]. An attacker can leverage these features to bypass any audit hooks or restricted environments, gaining unrestricted access to the underlying operating system and executing arbitrary machine code [84]. The `ctypes.pythonapi` provides a direct pathway to manipulate the interpreter's internals, effectively breaking out of any containment layer [84]. Consequently, attempting to run untrusted code from PyPI packages in a Python environment is inherently insecure. The recommended alternatives involve externalizing the sandboxing mechanism entirely, using technologies like container runtimes (gVisor, Kata Containers) or WebAssembly (WASI) runtimes, which provide stronger guarantees of resource confinement [53 84]. This admission by the language's own creators underscores a fundamental mismatch between Python's design philosophy and the requirements of a secure, multi-tenant, or collaborative neurotechnology platform.

Two specific architectural flaws stand out as particularly lethal in the context of AI and neurotechnology: insecure deserialization and dynamic code evaluation. The `pickle` module, a standard part of Python's serialization library, is notoriously insecure because it can execute arbitrary code during the deserialization process [85 86]. This capability stems from how Python objects are reconstructed; the pickle format stores not just data but also the fully qualified names of classes and functions, which are imported and executed upon unpickling [86]. An attacker who can influence the pickle stream can craft a payload that imports and executes any command, such as spawning a reverse shell [62 87]. This vulnerability is so severe that it has led to the deprecation of Django's `PickleSerializer` for session management [85]. The risk is amplified in the AI/ML domain, where models are often distributed as serialized objects. Attackers have successfully exploited this by embedding malicious payloads within PyTorch models, which are stored as zipped Pickle files, thereby delivering malware disguised as a legitimate machine learning asset [32 37 40].

Similarly, the `eval()` and `exec()` functions, which allow for the dynamic execution of code from strings, are powerful but inherently dangerous [63]. Their use with any form of untrusted input is a direct path to remote code execution [54 55]. Even if a developer attempts to restrict the scope of these functions by limiting the `__builtins__` namespace, reflection capabilities in Python allow attackers to circumvent these restrictions. By traversing an object's class hierarchy (e.g., `[x for x in (1).__class__.__base__.__subclasses__() ...]`), an attacker can "claw back" access to the original built-in functions and regain the ability to execute arbitrary commands [60]. This was demonstrated in the Yamale vulnerability, where an attacker could execute arbitrary code despite the framework's attempt to sanitize inputs passed to `eval()` [60]. Given that many web frameworks and data processing pipelines may construct code dynamically based on user input, the misuse of these functions creates a persistent and severe risk of code injection attacks [59 61].

Finally, from a systems engineering perspective, Python's performance characteristics are ill-suited for the real-time, deterministic control required by nanoswarming. The Global Interpreter Lock (GIL), while being phased out, historically prevented true parallelism on multi-core processors, limiting throughput for CPU-bound tasks [53]. More importantly, Python's garbage collection can introduce unpredictable pauses, causing latency spikes that are unacceptable in a closed-loop control

system where timing is critical [53]. Modern Python versions have improved this with incremental garbage collection, but the potential for non-deterministic behavior remains [53]. In contrast, compiled languages like Rust or Ada are designed for embedded and real-time systems, offering predictable, low-latency performance without the overhead of a managed runtime [21 89]. Perhaps most critically, high-assurance systems require mathematical proof of correctness—a concept known as formal verification. Languages like Ada, through its SPARK subset, are specifically designed to be amenable to formal methods, allowing developers to mathematically prove properties like the absence of runtime exceptions [23 24]. Python, with its dynamic typing and flexible object model, lacks the necessary structure to support this level of rigorous analysis, making it impossible to provide the same degree of assurance for its correctness and safety [26]. For a system governing nanoscale matter and human cognition, the absence of formal verification is a deal-breaker.

The Mismatch Between Python's Ecosystem and Emerging Neuro-Rights Frameworks

The deployment of Python in neuro-technology systems creates a profound and systemic conflict with the rapidly emerging global legal and ethical frameworks known as neurorights. Neurorights are proposed new human rights aimed at protecting individuals from the unprecedented threats posed by neurotechnologies that can monitor, interpret, and potentially alter brain activity [5 9]. These rights, centered on concepts like mental privacy, cognitive liberty, and mental integrity, are moving from academic discourse into binding legislation [3]. Chile, for instance, amended its constitution in 2021 to protect 'mental integrity,' and several U.S. states have enacted laws defining 'neural data' as a distinct category of protected personal information, requiring explicit consent for its collection and use [1 2 13]. The Python ecosystem, however, is fundamentally misaligned with the principles of these nascent rights. It lacks the inherent mechanisms for consent-by-design, offers no built-in protections for mental privacy, and perpetuates a governance vacuum where ethical considerations are an afterthought rather than a foundational requirement .

The core of the problem lies in the fact that Python-based tools and libraries are rarely designed with "consent-first" principles . Existing privacy laws already struggle to cover neural data, but the new neurorights frameworks impose stricter obligations, such as requiring express, informed, and specific consent before collecting or processing neural data [13 70]. A Neurorights Foundation audit of 30 consumer neurotechnology companies found that nearly all of them (96.7%) reserve the right to transfer brain data to third parties, fewer than 20% mention encryption, and only 10% adopt all core safety measures [8]. This systemic deficiency in corporate practices highlights the difficulty of achieving compliance in an unregulated environment. The Python ecosystem, which facilitates the rapid development of such applications, inherits this weakness. There is no mechanism within the language or its standard libraries to enforce granular, revocable consent. Data collection often occurs silently and without explicit user authorization, directly violating the spirit and letter of neurorights frameworks [28]. Without a "compliance-by-design" approach, which is antithetical to Python's culture of rapid prototyping, any system built on it is destined to operate in a state of perpetual non-compliance, exposing users to significant legal and ethical risks.

Furthermore, Python's architectural features actively undermine the right to mental privacy. The ease of data serialization and export means that sensitive datasets, including health information or personal identifiers, can be leaked with minimal lines of unmonitored code . Powerful introspection features like `getattr()` and `setattr()` can be exploited to hijack legitimate data-processing functions, enabling the covert capture and exfiltration of sensitive datasets without user knowledge . Debug and error output, if not rigorously managed in production, can unintentionally disclose sensitive details or system architecture, providing adversaries with valuable intelligence for further attacks [58]. These capabilities mean there is no inherent mechanism in the language to audit or prevent the unauthorized profiling, surveillance, or manipulation of individuals . This makes Python a poor choice for systems that must guarantee psychological continuity and protect against the erosion of mental privacy . The lack of strict authentication and input validation in many Python applications means users can be covertly profiled, surveilled, or manipulated, which is a direct violation of the principles underpinning neurorights .

This technological mismatch is exacerbated by the integration of artificial intelligence, which introduces a new layer of risk. Adversarial attacks can subtly manipulate AI models to degrade their accuracy, posing a direct threat to patient safety in medical applications [101]. For example, an adversarial attack could cause a diagnostic model to misclassify a neural pattern, leading to a wrong treatment decision [101]. Data poisoning attacks can embed backdoors into models during training, which can be triggered later to cause the system to malfunction or leak data [103,104]. Research has shown that injecting a tiny fraction of poisoned data into a training set can significantly increase the rate of harmful outputs from a language model, and these corrupted models remain harmful even after post-training mitigation attempts like fine-tuning [104]. Prompt injection attacks can manipulate large language models (LLMs) to ignore instructions and perform unauthorized actions, such as leaking sensitive data or accessing connected systems [103]. Since Python is the dominant language in the AI/ML space, its inherent weaknesses become a direct threat to the integrity of these advanced neuro-technological systems [19,101]. The EU's AI Act recognizes some of these dangers, prohibiting AI systems that use subliminal techniques to distort behavior, a category that includes potential manipulations facilitated by brain-machine interfaces [27].

To address this governance vacuum, a shift towards verifiable and decentralized architectures is necessary. Blockchain technology, with its inherent properties of immutability and traceability, offers a promising solution for ensuring accountability and enabling autonomous rollback [65,106]. By recording every action taken by the system in a tamper-proof ledger, blockchain can create an auditable chain of custody for all data and decisions, satisfying the need for forensic integrity [108,109]. This directly supports the enforcement of neurorights by providing a transparent record of data access and processing. Smart contracts can automate access controls and consent management, ensuring that data is only used in accordance with predefined rules [106]. Privacy-enhancing technologies (PETs) like federated learning and secure multi-party computation (SMPC) are also critical. These techniques allow for collaborative AI training without centralizing raw, sensitive neural data, thus respecting the principle of data minimization and preserving patient privacy by default [11,73,102]. Building a system on Python, which lacks these features natively, means that such protections must be bolted on as an afterthought, resulting in a fragile and less trustworthy system. To align with the Chimera Charter's

mandate for ethics-by-design, a new foundation must be built using technologies that are purpose-built for verifiability, decentralization, and privacy preservation.

Neuroright Principle	Definition	How Python Ecosystem Fails to Support It
Mental Privacy	Protection of thoughts, emotions, and mental experiences from unauthorized access or disclosure.	Lack of inherent mechanisms for consent auditing; easy data exfiltration via serialization/ introspection; verbose debug output leaks sensitive info.
Cognitive Liberty	Freedom from coercion and the right to self-determination over one's brain and mental experiences.	Absence of explicit consent frameworks; potential for covert profiling and manipulation via insecure data handling and AI models. [30 97]
Mental Integrity	The intactness and inviolacy of brain structure and function.	Memory safety vulnerabilities in C extensions can lead to unpredictable system behavior; insecure deserialization can allow arbitrary code execution, compromising system integrity. [14 30 86]
Psychological Continuity	The right to maintain a stable sense of self over time.	Lack of secure, auditable logging and rollback mechanisms; potential for data poisoning or adversarial attacks to corrupt learned models, altering future behavior. [65 103]
Consent & Control	Users must give clear, informed, and revocable consent for the collection and use of their neural data.	No built-in consent management; most tools do not respect or audit for informed consent; industry-wide failure to meet basic transparency standards. [90]

Systemic Governance Gaps and Environmental Vulnerabilities

The risks associated with Python extend beyond its language design and supply chain to encompass systemic governance gaps and inherent vulnerabilities in the collaborative environments where it is commonly used. These factors create a permissive atmosphere for abuse and compromise, making it difficult to establish a baseline of trust and security. The Python ecosystem's ethos of open collaboration, while beneficial for innovation, lacks the robust governance structures needed to police unethical behavior or enforce compliance with emerging legal standards like neurorights. This results in a fragmented and often chaotic landscape where individual developers and small teams bear the brunt of securing their own tools and dependencies, leaving large-scale infrastructure vulnerable. Furthermore, the very environments that facilitate rapid development and data science, such as Jupyter notebooks and cloud-based services like Google Colab, inherit and amplify Python's risks, creating attractive targets for attackers.

A significant systemic gap is the lack of a centralized authority or framework for embedding neuro-rights, cognitive liberty, or constitutional protections into the tooling or package metadata of the

Python ecosystem . Unlike regulated industries where standards bodies dictate security and privacy requirements, the open-source world operates on a consensus-driven model that prioritizes functionality and developer convenience. As a result, ethical considerations are frequently absent from the design process, leading to tools that are powerful but dangerously naive [28]. The Neurorights Foundation's findings that 96.7% of consumer neurotech companies reserve the right to transfer neural data to third parties demonstrates a systemic industry-wide failure to prioritize user control and privacy [8]. The Python ecosystem, by providing the primary software stack for this industry, becomes complicit in perpetuating this status quo. Without a mandated "ethics-by-design" framework, there is no incentive for package maintainers to implement features like granular consent management, data minimization, or secure deletion protocols, leaving these critical safeguards to be implemented ad hoc and inconsistently by end-users [28].

This governance vacuum is starkly illustrated by the behavior of major technology companies. Apple's filing of a patent for emotion tracking via sensors in future AirPods and Facebook's internal workshops on using EEG devices to optimize newsfeeds reveal a corporate ambition to monetize neural data, even as they publicly defer on ethical concerns [69]. Companies like NeuroSky openly state in their privacy policies that they can share or sell user brain data to any number of undisclosed partners [69]. This commercial exploitation of neural data stands in direct opposition to the principles of neurorights, which seek to treat brain data as a special category of information requiring the highest level of protection [68 97]. The Python ecosystem, with its emphasis on rapid development cycles and open APIs, provides the ideal conduit for these data flows, facilitating the collection, processing, and sharing of sensitive information with minimal friction. The lack of strong authentication and input validation in many Python applications further enables this, allowing for covert profiling and surveillance that violates the core tenets of mental privacy and cognitive freedom .

Collaborative development environments, while indispensable for modern data science, introduce their own set of acute security risks. Cloud notebook environments like Jupyter and Google Colab are widely used in neuroscience research and machine learning development [19]. However, they inherit Python's risks and amplify them by providing a shared workspace where credential theft can occur with minimal obfuscation [105]. A malicious actor could easily inject a script into a shared notebook that exfiltrates environment variables, browser cookies, or other sensitive credentials upon execution [105]. The arbitrary code execution capabilities of Python mean that a single compromised cell in a notebook could grant an attacker full control over the entire computational environment, including any attached cloud storage or database connections [105]. The incident where a malicious Python package was used to demonstrate arbitrary code execution in Google Colab highlights the danger of these platforms [105]. For a system like Project Chimera, which handles highly sensitive neuro-data, relying on such environments for development or data analysis is untenable, as it exposes the entire system to a high-risk attack surface.

Perhaps the most insidious environmental vulnerability is the practice of automated and silent updates. Many Python environments rely on automated dependency updates managed by tools like `pip` . While convenient, this practice can be a vector for delivering malicious code to millions of systems without user awareness or control . If a malicious package is pushed to PyPI and a project's dependency file specifies a broad range or the latest version, an automated update could silently

install the compromised version, triggering a supply chain attack at scale . This bypasses any manual review process and undermines the principle of least privilege. The risk is magnified in enterprise settings where thousands of interconnected systems depend on a common set of libraries. A single vulnerable or malicious upstream dependency can propagate through complex project trees, enabling mass-scale data theft or system compromise across entire organizations or global research consortia . This creates a systemic risk where the security of a single, poorly maintained package can have cascading consequences for the entire ecosystem. This reliance on a constantly changing and often unvetted set of dependencies makes it incredibly difficult to establish a secure and immutable baseline for a critical system, a cornerstone of high-assurance engineering.

Viable Alternatives and a Secure Architecture for Mission-Critical Nanoswarms

Given the profound and multifaceted risks inherent in the Python ecosystem, the conclusion is unequivocal: for mission-critical nanoswarm systems, Python is an unacceptable technical foundation. Its design priorities of developer productivity and ecosystem richness are fundamentally misaligned with the non-negotiable demands of absolute security, real-time deterministic control, and unwavering commitment to neuro-rights and constitutional compliance [16]. The solution is not to continue mitigating Python's shortcomings, which are architectural and systemic in nature, but to build a new foundation using a technology stack and architectural paradigm designed from the ground up for safety, verifiability, and resilience. This requires a deliberate shift away from interpreted, dynamically typed languages toward compiled, statically analyzable, and formally verifiable alternatives, combined with a secure-by-design architectural approach that incorporates advanced technologies for isolation, integrity, and privacy.

The most suitable alternative languages are memory-safe systems programming languages like Rust and Ada. These languages are engineered to prevent entire classes of vulnerabilities that plague C/C++ and, by extension, Python's native extensions [88 89]. Rust, in particular, achieves memory safety at compile time through its ownership and borrowing system, eliminating buffer overflows, use-after-free, and other common memory errors without relying on a garbage collector [89]. This is critical for safety-critical systems where a memory corruption bug could lead to catastrophic failure [80]. Rust also provides deterministic, low-latency performance, making it ideal for the real-time control loops required by nanoswarms [89]. Crucially, the Rust ecosystem includes certified toolchains like Ferrocene, which are compliant with stringent industry standards for safety-critical software, such as IEC 62304 for medical devices and DO-178C for aerospace [88 89]. This certification provides a clear path to regulatory approval and builds confidence in the system's reliability. Ada, another language designed for high-integrity systems, offers similar guarantees of memory safety and predictability, and its SPARK subset is specifically designed to support formal verification, allowing developers to mathematically prove critical program properties [23 24]. These languages represent a paradigm shift from Python's "dynamic and hope for the best" approach to a "statically verified and guaranteed" methodology.

Building on a secure language foundation is only part of the solution. A truly resilient architecture must incorporate multiple layers of defense. First, the system should be architected around secure

edge computing principles, processing sensitive neural data locally on-device whenever possible to minimize transmission-related risks and reduce reliance on network connectivity [41]. Second, hardware-based trusted execution environments (TEEs), such as Intel SGX, should be utilized for isolating the most sensitive computations [42 77]. TEEs provide a hardware-enforced secure enclave where code and data can be processed in complete isolation from the main operating system, protecting them from even privileged attackers [42]. Third, to address the need for accountability and autonomous rollback, a permissioned blockchain should be integrated as the system's immutable ledger [65 106]. Every critical action, data access, and configuration change would be recorded as a transaction on the blockchain, creating a tamper-proof audit trail that satisfies regulatory requirements and enables forensic investigation [108 109]. This directly counteracts the lack of verifiability in Python-based systems.

Finally, to ensure the system respects the core tenets of neurorights, Privacy-Enhancing Technologies (PETs) must be woven into the fabric of the architecture. Federated learning, for example, allows for collaborative AI model training without ever centralizing raw patient data, thus preserving data sovereignty and minimizing privacy risks [11 73 102]. Homomorphic encryption can enable computations to be performed directly on encrypted data, while secure multi-party computation (SMPC) allows multiple parties to jointly compute a function over their inputs without revealing those inputs to each other [11 73]. These technologies are essential for building a system that can innovate and learn from data without compromising the fundamental right to mental privacy. By adopting this holistic approach—combining a memory-safe language like Rust, a secure edge and TEE architecture, a blockchain for auditability, and PETs for privacy—the Chimera project can move beyond simply avoiding Python's pitfalls and instead construct a system that is genuinely secure, verifiable, and ethically aligned from its inception.

Feature	Unsuitable: Python	Suitable Alternative: Rust / Ada	Rationale for Adoption
Memory Safety	Generally safe in pure Python, but unsafe via C extensions. Prone to memory corruption bugs.	Compile-time guarantees prevent buffer overflows, use-after-free, etc. No garbage collector overhead.	Eliminates ~70% of critical software vulnerabilities at their source, essential for safety-critical systems. [76 80 89]
Runtime Isolation	Not designed to be a secure sandbox; dynamic features allow breakout.	Built-in type system prevents many unsafe operations. Integrates with WebAssembly (WASI) for strong sandboxing.	Provides a reliable foundation for running untrusted code or enforcing strict access controls between modules. [53 84]
Deterministic Performance	Garbage collection can cause unpredictable pauses; GIL limits parallelism.	Predictable, low-latency performance with no runtime overhead or	Essential for real-time control and coordination of nanoswarming agents where timing is critical. [53 89]

Feature	Unsuitable: Python	Suitable Alternative: Rust / Ada	Rationale for Adoption
		pauses. Supports hard-real-time programming.	
Verifiability	Lacks strong typing and formal methods support; impossible to mathematically prove safety/correctness.	Supports formal verification (SPARK Ada) and extensive static analysis. Strong, optional type system.	Enables mathematical proof of correctness, meeting the highest standards of engineering integrity for life-critical systems. [24] [26] [88]
Supply Chain Security	Open ecosystem with no mandatory review; high risk of malicious packages.	Growing ecosystem with tools like Cargo for generating SBOMs and verifying dependencies.	Reduces reliance on untrusted public repositories through better dependency management and transparency. [80] [89]
Ethical Alignment	No inherent mechanisms for consent, privacy, or auditability.	Can be combined with blockchain and PETs to build systems that are privacy-preserving and accountable by design.	Directly supports the "ethics-by-design" mandate of the Chimera Charter, enabling compliance with neurorights frameworks. [73] [106]

Reference

1. Neurorights in the Constitution: from neurotechnology to ethics ... https://pmc.ncbi.nlm.nih.gov/articles/PMC11491849/
2. Neural Data Privacy Regulation: What Laws Exist and ... https://www.arnoldporter.com/en/perspectives/advisories/2025/07/neural-data-privacy-regulation
3. Mind the Gap: Lessons Learned from Neurorights https://www.sciencediplomacy.org/article/2022/mind-gap-lessons-learned-neurorights
4. Neuroethics: What's Next for Regulating Neurotechnology? https://www.neuroelectrics.com/blog/neuroethics-what-s-next-for-regulating-neurotechnology
5. Neurorights: Is the creation of new human rights effective in ... https://www.ibanet.org/neurorights-human-dignity
6. Neurorights Foundation: Home | NRF https://www.neurorightsfoundation.org/
7. Mapping neurotech governance https://cfg.eu/neurotech-governance-map/

8. Neurotechnology Privacy: Safeguarding the Next Frontier ... https://trustarc.com/resource/neurotechnology-privacy-safeguarding-the-next-frontier-of-data/
9. It's Time for Neuro - Rights https://www.cirsd.org/en/horizons/horizons-winter-2021-issue-no-18/its-time-for-neuro--rights
10. Neurotechnology and neurorights - Privacy's last frontier https://www.europarl.europa.eu/thinktank/de/events/details/neurotechnology-and-neurorights-privacy-/20231019WKS05721
11. Protecting Privacy of Users in Brain-Computer Interface ... https://www.researchgate.net/publication/334286648_Protecting_Privacy_of_Users_in_Brain-Computer_Interface_Applications
12. Navigating the legal and ethical landscape of brain- ... https://iapp.org/news/a/navigating-the-legal-and-ethical-landscape-of-brain-computer-interfaces-insights-from-colorado-and-minnesota
13. Unlocking Neural Privacy: The Legal and Ethical Frontiers ... https://www.cooley.com/news/insight/2025/2025-03-13-unlocking-neural-privacy-the-legal-and-ethical-frontiers-of-neural-data
14. (PDF) Ethical and Safety Challenges of Implantable Brain- ... https://www.researchgate.net/publication/391482749_Ethical_and_Safety_Challenges_of_Implantable_Brain-Computer_Interface
15. Brain − computer interface: trend, challenges, and threats - PMC https://pmc.ncbi.nlm.nih.gov/articles/PMC10403483/
16. Brain-Computer Interfaces: Applications, Challenges, and ... https://www.gao.gov/products/gao-25-106952
17. Unpacking the Invisible Threat: How Brain-Computer ... https://breached.company/unpacking-the-invisible-threat-how-brain-computer-interfaces-can-be-hacked/
18. A blockchain security module for brain-computer interface ... https://www.sciencedirect.com/science/article/pii/S2772528621000303
19. Evaluating the real-world usability of BCI control systems ... https://www.frontiersin.org/journals/human-neuroscience/articles/10.3389/fnhum.2024.1448584/full
20. Neurodata Consent Frameworks: Managing EEG/Brain ... https://secureprivacy.ai/blog/neurodata-consent-eeg-brain-computer-interface-data-gdpr-ccpa
21. Programming Languages in Safety-Critical Applications https://www.adesso.de/en/news/blog/programming-languages-in-safety-critical-applications.jsp
22. Formal Methods for Secure Software Design in Safety-Critical ... https://willbates1.medium.com/formal-methods-for-secure-software-design-in-safety-critical-systems-1d7d8a0fd5ff
23. Formal Verification of Safety Critical Software in Ada https://dl.acm.org/doi/10.1145/3742939.3742943
24. SAFETY CRITICAL DESIGN https://www.adacore.com/uploads/techPapers/Safety-Critical_Design.pdf

25. Formal methods in C++ for safety critical software https://stackoverflow.com/questions/24899711/formal-methods-in-c-for-safety-critical-software
26. Practical Verification of Safety-Critical Systems https://digital.lib.washington.edu/researchworks/items/fa2d685b-29a1-4dac-85c1-80e1bde12007
27. Neurotechnologies under the EU AI Act: Where law meets ... https://iapp.org/news/a/neurotechnologies-under-the-eu-ai-act-where-law-meets-science
28. Four ethical priorities for neurotechnologies and AI - PMC https://pmc.ncbi.nlm.nih.gov/articles/PMC8021272/
29. 'Neurorights' (Chapter 24) - The Cambridge Handbook of ... https://www.cambridge.org/core/books/cambridge-handbook-of-responsible-artificial-intelligence/neurorights/AF85DE57D51D114E26C19146E234F897
30. Mapping Ethical and Legal Foundations of 'Neurorights' https://scholarship.law.duke.edu/cgi/viewcontent.cgi?article=7079&context=faculty_scholarship
31. PYPI Security: How to Prevent Supply Chain Attacks ... https://bolster.ai/blog/pypi-supply-chain-attacks
32. Malicious PyPI, npm, and Ruby Packages Exposed in ... https://thehackernews.com/2025/06/malicious-pypi-npm-and-ruby-packages.html
33. GitLab uncovers Bittensor theft campaign via PyPI https://about.gitlab.com/blog/gitlab-uncovers-bittensor-theft-campaign-via-pypi/
34. Security Risks with Python Package Naming Convention https://snyk.io/articles/security-risks-with-python-package-naming-convention-typosquatting-and/
35. Securing the Software Supply Chain from Typosquatting ... https://www.optiv.com/insights/discover/blog/securing-software-supply-chain-typosquatting-attacks
36. Malicious PyPI Packages Deliver SilentSync RAT https://www.zscaler.com/blogs/security-research/malicious-pypi-packages-deliver-silentsync-rat
37. Multiple Malicious Packages Discovered on PyPI, npm, ... https://cloudsmith.com/blog/multiple-malicious-packages-discovered-on-pypi-npm-and-rubygems
38. Finding malicious PyPI packages through static code ... https://securitylabs.datadoghq.com/articles/guarddog-identify-malicious-pypi-packages/
39. Malicious Package Detection in NPM and PyPI using a ... https://dl.acm.org/doi/10.1145/3705304
40. Malicious attack method on hosted ML models now targets ... https://www.reversinglabs.com/blog/malicious-attack-method-on-hosted-ml-models-now-targets-pypi
41. Secure Edge Computing: Transforming IoT in Healthcare ... https://www.iotcream.com/2024/02/20/secure-edge-computing-transforming-iot-in-healthcare-manufacturing-and-beyond/

42. Secure Computing Enclaves Move Digital Medicine Forward https://www.mayoclinicplatform.org/2021/09/09/secure-computing-enclaves-move-digital-medicine-forward/
43. How To Build a Safe, Secure Medical AI Platform | Stanford HAI https://hai.stanford.edu/news/how-to-build-a-safe-secure-medical-ai-platform
44. Sigstore Information https://www.python.org/downloads/metadata/sigstore/
45. PyPI's Sigstore-powered attestations are now generally ... https://blog.sigstore.dev/pypi-attestations-ga/
46. PEP 740 – Index support for digital attestations https://peps.python.org/pep-0740/
47. Python and Sigstore https://sethmlarson.dev/python-and-sigstore
48. Python https://docs.sigstore.dev/language_clients/python/
49. PEP 761 – Deprecating PGP signatures for CPython artifacts https://peps.python.org/pep-0761/
50. Attestations: A new generation of signatures on PyPI https://blog.trailofbits.com/2024/11/14/attestations-a-new-generation-of-signatures-on-pypi/
51. PyPI now supports digital attestations https://blog.pypi.org/posts/2024-11-14-pypi-now-supports-digital-attestations/
52. PyCon 2023 https://yossarian.net/res/pub/pycon-2023.pdf
53. Python 3.14 – What you need to know https://cloudsmith.com/blog/python-3-14-what-you-need-to-know
54. The Dark Side of Python's eval() (and When It's Actually ... https://leapcell.io/blog/python-eval-how-it-works-and-why-its-risky
55. Command Injection in Python: Examples and Prevention https://www.stackhawk.com/blog/command-injection-python/
56. Hackers Abuse Python eval/exec Calls to Run Malicious ... https://gbhackers.com/hackers-abuse-python-eval-exec-calls/
57. Why should exec() and eval() be avoided? - python https://stackoverflow.com/questions/1933451/why-should-exec-and-eval-be-avoided
58. 5 Python Security Traps You Need to Avoid https://blogs.cisco.com/developer/pythonvulnerabilities01
59. 94 Code Injection in python_executor Class Due to ... https://github.com/GAIR-NLP/factool/issues/50
60. Newly discovered code injection vulnerability in Yamale https://jfrog.com/blog/23andmes-yamale-python-code-injection-and-properly-sanitizing-eval/
61. Code injection in Python: examples and prevention https://snyk.io/blog/code-injection-python-prevention-examples/

62. Hacking Python Applications https://medium.com/swlh/hacking-python-applications-5d4cd541b3f1

63. The dark side of Python: Why eval() and exec() are security ... https://python.plainenglish.io/the-dark-side-of-python-why-eval-and-exec-are-security-nightmares-36b9406dc140

64. Nanotechnology in healthcare, and its safety and ... https://jnanobiotechnology.biomedcentral.com/articles/10.1186/s12951-024-02901-x

65. (PDF) Output-Valid Rollback-Recovery https://www.researchgate.net/publication/228788109_Output-Valid_Rollback-Recovery

66. Swarm Learning for decentralized and confidential clinical ... https://www.nature.com/articles/s41586-021-03583-3

67. Privacy-Aware Cloud Architecture for Collaborative Use of ... https://www.mdpi.com/2076-3417/13/13/7401

68. Mental Privacy in Neurotech and the Growing Risk for ... https://verasafe.com/blog/mental-privacy-in-neurotech-and-the-growing-risk-for-organizations/

69. The Need for Expanded Legal Protections of Brain Data https://www.researchgate.net/publication/380493553_Safeguarding_Neural_Privacy_The_Need_for_Expanded_Legal_Protections_of_Brain_Data

70. How to build a secure neural data framework with privacy https://insights.manageengine.com/privacy-compliance/neural-data-privacy/

71. Neural, Reproductive, and Other Sensitive Health Data https://www.privacysecurityacademy.com/neural-reproductive-and-other-sensitive-health-data-privacy-risks-and-transparency-responsibilities/

72. Neural data privacy an emerging issue as California signs ... https://epic.org/the-record-neural-data-privacy-an-emerging-issue-as-california-signs-protections-into-law/

73. A Privacy-Preserving and Attack-Aware AI Approach for ... https://www.mdpi.com/2079-9292/14/7/1385

74. Towards understanding the security issues of Python ... https://dl.acm.org/doi/full/10.1145/3755881.3755886

75. 5 Common Memory Bugs and How to Secure Your Systems https://sternumiot.com/iot-blog/memory-safety-5-common-memory-bugs-and-how-to-secure-your-system/

76. Is Python memory-safe? https://stackoverflow.com/questions/61974312/is-python-memory-safe

77. Towards Memory Safe Python Enclave for Security ... https://arxiv.org/pdf/2005.05996

78. How Performance Became the Nemesis of the Secure ... https://www.securityjourney.com/post/how-performance-became-the-nemesis-of-the-secure-python-code

79. This memory safety issue everyone's talking about... It's not ... https://www.reddit.com/r/cpp/comments/16uwauu/this_memory_safety_issue_everyones_talking_about/

80. The Case for Memory Safe Roadmaps https://media.defense.gov/2023/Dec/06/2003352724/-1/-1/0/THE-CASE-FOR-MEMORY-SAFE-ROADMAPS-TLP-CLEAR.PDF

81. CVE-2024-12254: Python Memory Exhaustion High ... https://linuxsecurity.com/news/security-vulnerabilities/new-python-memory-exhaustion-bug

82. NSA and CISA Urge Adoption of Memory Safe Languages ... https://www.infosecurity-magazine.com/news/nsa-cisa-urge-memory-safe-languages/

83. The Ultimate Guide to Open-Source Security with Python and R https://www.anaconda.com/guides/the-ultimate-guide-to-open-source-security-with-python-and-r

84. Built-in security model https://discuss.python.org/t/built-in-security-model/66572

85. Insecure Deserialization in Python https://semgrep.dev/docs/learn/vulnerabilities/insecure-deserialization/python

86. pickle — Python object serialization https://docs.python.org/3/library/pickle.html

87. Python Serialization Vulnerabilities - Pickle https://www.hackingarticles.in/python-serialization-vulnerabilities-pickle/

88. Rust for Medical Devices: Secure & Certified Software ... https://yalantis.com/blog/rust-for-medical-devices/

89. Rust for Embedded and Safety-Critical ... https://ferrous-systems.com/pdf/rust-for-embedded-safety-critical-systems-2025.pdf

90. Safeguarding Brain Data: Assessing the Privacy Practices ... https://perseus-strategies.com/wp-content/uploads/FINAL_Consumer_Neurotechnology_Report_Neurorights_Foundation_April-1.pdf

91. States Pass Privacy Laws To Protect Brain Data Collected ... https://kffhealthnews.org/news/article/colorado-california-montana-states-neural-data-privacy-laws-neurorights/

92. Defining “Neural Data” in U.S. State Privacy Laws https://fpf.org/blog/the-neural-data-goldilocks-problem-defining-neural-data-in-u-s-state-privacy-laws/

93. States Pass Privacy Laws Safeguarding Brain Data Collected https://www.govtech.com/policy/states-pass-privacy-laws-safeguarding-brain-data-collected

94. A new law in California protects consumers' brain data. ... https://www.technologyreview.com/2024/10/04/1104972/law-california-protects-brain-data-doesnt-go-far-enough/

95. Privacy violations top incidents related to AI, report finds https://www.hcamag.com/us/news/general/privacy-violations-top-incidents-related-to-ai-report-finds/546262

96. U.S. States Move to Protect Brain Data Collected from ... https://quaylogic.com/u-s-states-move-to-protect-brain-data-collected-from-consumer-devices/

97. 'Neurorights' and the next flashpoint of medical privacy https://iapp.org/news/a/an-introduction-to-neurorights-and-the-next-flashpoint-of-medical-privacy
98. Neural Data Privacy https://legis.delaware.gov/docs/default-source/publications/issuebriefs/neuraldataprivacyissuebrief.pdf?sfvrsn=10df8838_1
99. AI Security: Risks, Frameworks, and Best Practices https://www.wallarm.com/what/ai-security-risks-frameworks-and-best-practices
100. 7 AI Security Risks (+ Modern Mitigation Strategies) https://svitla.com/blog/common-ai-security-risks/
101. (PDF) Network Security in AI-based healthcare systems https://www.researchgate.net/publication/387205582_Network_Security_in_AI-based_healthcare_systems
102. Privacy-preserving artificial intelligence in healthcare https://www.sciencedirect.com/science/article/pii/S001048252300313X
103. Securing AI Systems: A Guide to Known Attacks and Impacts https://arxiv.org/html/2506.23296v1
104. Medical large language models are vulnerable to data ... https://pmc.ncbi.nlm.nih.gov/articles/PMC11835729/
105. Cyber security risks to artificial intelligence https://www.gov.uk/government/publications/research-on-the-cyber-security-of-ai/cyber-security-risks-to-artificial-intelligence
106. The Application of Blockchain Technology in the Field ... https://www.mdpi.com/2813-5288/3/1/5
107. Malicious Python Packages Exploit Popular ... https://cyberpress.org/malicious-python-packages-exploit-popular-cryptocurrency-library/
108. Collaborative penetration testing suite for emerging ... https://pmc.ncbi.nlm.nih.gov/articles/PMC12532622/
109. A Framework for Blockchain-Based Access Logs and ... https://www.researchgate.net/publication/392312120_A_Framework_for_Blockchain-Based_Access_Logs_and_Tamper-Proof_Audit_Trails
110. AI and Blockchain Integration - Serokell https://serokell.io/blog/ai-blockchain-integration
111. Malicious Python packages target popular Bitcoin library https://www.reversinglabs.com/blog/malicious-python-packages-target-popular-bitcoin-library
112. Collaborative penetration testing suite for emerging ... https://link.springer.com/article/10.1007/s10489-025-06908-1
113. Blockchain Audit Trails: Revolutionizing Enterprise ... https://www.myshyft.com/blog/blockchain-for-audit-trails/
114. rkalis/blockchain-audit-trail: Demo application ... https://github.com/rkalis/blockchain-audit-trail