Architectural Specification and Security Evaluation of the Secure Asynchronous Full-Duplex Stream Framework (SAFD-FRAMEWORK-SPEC-01)

1. Request for Comments (RFC): Abstract & Threat Directives

1.1 Abstract

Managed application runtimes that rely on top-to-bottom script re-execution models, such as Streamlit, present severe concurrency and security challenges when hosting persistent, real-time networking protocols[1]. Standard client WebSockets implementations operating under RFC 6455 rely on long-lived event loops that block the executing thread while waiting for frame arrivals or processing backpressure[2]. When introduced directly into a script-reexecution runtime, these blocking loops lock the primary user interface thread, resulting in interface starvation, unresponsiveness, and state corruption upon user input triggers[1].

Simultaneously, validating external target URIs within managed application environments introduces critical application security risks[5]. Relying on perimeter network security devices or standard operating system DNS resolution exposes the application layer to Server-Side Request Forgery (SSRF), Time-of-Check to Time-of-Use (TOCTOU) DNS Rebinding attacks, and internal network pivoting[4]. Hostnames evaluated as safe during pre-connection checks can re-resolve to restricted loopback or internal address spaces during the socket connection phase[4]. Furthermore, transition technologies designed to map IPv4 protocols over IPv6 infrastructure can bypass standard Subnet CIDR blocklists if the application engine fails to unpack embedded IPv4 payloads hidden within IPv6 interface identifiers[5].

The Secure Asynchronous Full-Duplex Stream Framework (SAFD-FRAMEWORK-SPEC-01) addresses these security and operational challenges through an integrated design. By isolating DNS resolution inside a synchronized execution pool, enforcing direct-IP literal pinning, recursively unwrapping subterranean IPv6 transition headers, and decoupling the WebSocket event loop into a dedicated OS worker thread connected via bounded inter-thread queues, SAFD provides continuous full-duplex communication without sacrificing perimeter security or interface responsiveness[3].

1.2 Threat Vectors & Security Directives

Standard network socket initializations inherently trust operating system resolver libraries, creating significant windowing vulnerabilities[3]. In a classic DNS Rebinding scenario, an adversary configures an authoritative name server to issue responses with minimal or zero

Time-To-Live (TTL) values[4]. An initial query yields a benign public IP address that passes validation logic; however, when the underlying network client establishes the TCP connection, the OS executes a second lookup that returns a restricted address such as 127.0.0.1 or 169.254.169.254[4].

To neutralize TOCTOU DNS Rebinding vectors completely, SAFD enforces pre-resolved IP pinning[3]. Handshake URIs undergo strict pre-processing in an isolated thread pool using synchronous resolution functions bounded by a hard timeout[3]. Once validated against security policies, the original hostname in the URI target string is overwritten by the IP literal[3]. To preserve virtual host routing and TLS SNI validation without triggering secondary DNS lookups, the original hostname is explicitly passed through the HTTP Host header and the TLS server_hostname context parameter[3].

A secondary, highly prevalent vector involves subterranean IPv6 SSRF tunneling[5]. Standard validation tools evaluate network addresses using shallow IPv6 string or prefix comparisons, failing to inspect embedded IPv4 target payloads[5]. An attacker can supply a routable IPv6 endpoint that secretly encapsulates a private IPv4 target address via transition protocols such as 6to4, Teredo, NAT64, or ISATAP[5]. Upon reaching an edge gateway, the encapsulated packet is decrypted or translated, pivoting into internal corporate assets[5]. SAFD mandates recursive binary unwrapping of all IPv6 addresses to extract embedded IPv4 targets prior to passing boundary evaluation[5].

Finally, placing blocking socket event loops inside script rendering threads causes UI thread starvation and unmitigated memory expansion[1]. SAFD isolates socket I/O into a dedicated OS worker thread running a fresh event loop[10]. Inter-thread datagram exchange occurs exclusively over bounded queue primitives, protected by non-blocking operations and UI fragment polling to enforce strict memory bounds and preserve rendering loop execution[3].

2. Ingress Validation & IPv6 Subterranean Inspection Mechanics

2.1 Direct-IP Pinning & Header Mapping Execution Sequence

The ingress security engine executes a multi-stage validation sequence prior to returning an approved connection handle. The process begins when a user submits a candidate URI to the application runtime layer. The validation engine immediately screens the input string for control character injection, rejecting any URI containing carriage returns (\r), line feeds (\n), or null bytes (\0) to prevent HTTP header splitting and protocol smuggling attacks[2]. The URI scheme is then validated, enforcing a strict allowlist of ws or wss[2]. Embedded userinfo credentials (e.g., ws://user:pass@host) are explicitly prohibited to prevent credential leakage through downstream logging or proxy handling[3].

Once the initial URI syntax passes evaluation, the engine parses the target hostname and port[3]. If the hostname is provided as a raw IP address, it is immediately converted to an address object for boundary inspection. If the host is expressed as a domain name, resolution is offloaded to a ThreadPoolExecutor running socket.getaddrinfo() bounded by a DNS_TIMEOUT

limit of 3.0 seconds[3]. This isolation prevents slow or malicious DNS servers from exhausting main application threads.

When DNS resolution returns multiple IP address records, every resolved candidate address must pass the strict subnetwork inspection routine. If any returned IP maps to a restricted space, the entire connection request is denied. Upon successful verification of a public IP address, the engine constructs a pinned URI string by replacing the target domain with the IP literal, enclosing IPv6 addresses in square brackets[3]. To ensure compliance with standard web security models, the original domain string is retained in a metadata object and explicitly injected into the HTTP Host header and the TLS server_hostname handshake parameter[3].

2.2 Recursive Subterranean IPv6 Payload Unwrapping Logic

To counter subterranean SSRF bypass attempts, the SAFD validation framework inspects every resolved IP address using bitwise unwrapping routines prior to checking subnet blocklists[5]. When an IPv6 address is evaluated, the validation engine checks for the presence of embedded IPv4 payloads across five distinct transition mechanisms[5]:

1. **IPv4-Mapped IPv6 Addresses (::ffff:0:0/96):** The lower 32 bits directly encode an IPv4 target[5]. The unwrapper extracts the ip.ipv4_mapped property to yield the underlying native IPv4 address object[5].
2. **6to4 Tunneling Infrastructure (2002::/16):** Standardized under RFC 3056, 6to4 encapsulates IPv4 packets across IPv6 networks[5]. Octets 2 through 5 of the binary packed representation contain the embedded IPv4 destination, extracted via slice ip.packed[2:6][5].
3. **NAT64 Well-Known Prefixes (64:ff9b::/96):** Standardized under RFC 6052 for IPv6-to-IPv4 translation[5]. The target IPv4 destination occupies the trailing 32 bits, extracted via slice ip.packed[12:16][5].
4. **Teredo NAT Traversal (2001::/32):** Standardized under RFC 4380, Teredo encapsulates IPv6 datagrams inside IPv4 UDP packets[5]. The client IPv4 locator resides in octets 12 through 15 but is obfuscated via a bitwise XOR operation against a fixed mask of 0xFFFFFFFF[5]. The mathematical extraction formula is expressed as:

$$\text{IPv4}_{\text{raw}} = \text{PackedBytes}[12 : 16] \oplus \text{0xFFFFFFFF}$$

5. **ISATAP Interface Identifiers (RFC 5214):** The Intra-Site Automatic Tunnel Addressing Protocol embeds IPv4 locators inside the 64-bit interface identifier of an IPv6 address[7]. An ISATAP address uses the specific 16-bit marker 0x5EFE in octets 8 and 9 (formatted as 0:5efe: or 200:5efe: depending on address global uniqueness)[8]. The extraction engine isolates octets 12 through 15 (ip.packed[12:16]) to inspect the embedded IPv4 host address[15].

If an embedded IPv4 address is extracted from any of these transition mechanisms, the resulting IPv4 object is passed recursively back into the boundary inspector to ensure

complete compliance against IPv4 blocklists[5].

2.3 Strict Subnet Boundary Enforcement Matrix

Following bitwise extraction, the final IPv4 or IPv6 address object is evaluated against standard system blocklists and explicit subnetwork boundaries[5]. Access is rejected if the target address matches any entry in the following boundary matrix:

Subnet Identifier	CIDR Range	Security Rationalization & Risk Profile
Private Networks	10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16	Prevents internal network pivoting to corporate intranets and private microservices (RFC 1918)[5].
Loopback Interfaces	127.0.0.0/8, ::1/128	Mitigates local service exploitation (e.g., Redis, local administrative REST endpoints)[5].
Link-Local Addresses	169.254.0.0/16, fe80::/10	Blocks unauthorized access to cloud platform instance metadata services (IMDSv1/IMDSv2)[5].
Carrier-Grade NAT	100.64.0.0/10	Restricts routing into shared service provider infrastructure and CGNAT spaces (RFC 6598).
Unique Local Addresses	fc00::/7	Prevents unauthorized connection attempts to isolated IPv6 private enterprise networks (RFC 4193).
Documentation Ranges	192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24,	Blocks routing to non-operational test networks (TEST-NET-1,

	2001:db8::/32	TEST-NET-2, TEST-NET-3).
Broadcast & Reserved	0.0.0.0/8, 255.255.255.255/32	Rejects wildcard bindings, local subnet broadcasts, and unassigned network segments.

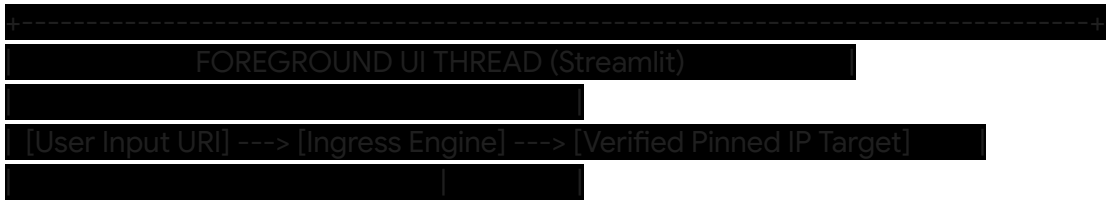
3. System Design Paper: Concurrency Pipeline & Structural Comparison

3.1 Problem Statement in Script-Reexecution Environments

Managed application environments like Streamlit operate on an execution paradigm where user interactions trigger a complete, top-to-bottom re-execution of the Python script on the primary user interface thread. Hosting a long-lived, synchronous, or blocking asynchronous WebSocket loop directly within this execution path presents two major systemic failures:
First, maintaining a blocking socket read loop (ws.recv()) halts script execution, preventing the runtime from processing UI events, rendering updated layout components, or acknowledging user inputs[1]. The application interface freezes until the socket connection closes or encounters a catastrophic read timeout.
Second, relying exclusively on perimeter proxies for SSRF protection leaves application runtimes vulnerable if users can input arbitrary URIs[4]. Modern microservice architectures frequently allow egress connections from application containers to external WebSockets[13]. Without application-layer IP pinning and transition-aware validation, an attacker can input internal endpoints or tunneling URIs, bypassing edge network controls and executing SSRF pivots directly from the application host[4].

3.2 Full-Duplex Dataflow Architecture

The SAFD architecture resolves thread starvation and security vulnerabilities by completely separating network execution from UI rendering. The dataflow operates across a strict boundary dividing the foreground Streamlit main thread from an isolated background OS worker thread[10].





The execution flow progresses systematically:

1. **User URI Submission & Ingress Validation:** The user submits a target URI through the foreground UI layer. The main thread passes the string to the ingress validation engine, which executes DNS resolution inside a temporary thread pool, performs subterranean IPv6 unwrapping, checks subnet blocklists, and builds a pinned IP URI metadata structure[3].
2. **Thread Instantiation & Loop Setup:** Upon successful validation, the main thread instantiates a dedicated OS worker thread (threading.Thread), passing the pinned URI, metadata, inter-thread queues, and a shutdown event object[10]. The worker thread creates a fresh asyncio event loop using asyncio.new_event_loop(), sets it as the active loop for that thread, and starts socket execution[10].
3. **Full-Duplex Socket Management:** Inside the background worker thread, websockets.connect() establishes a connection to the pinned IP literal while setting the Host header and server_hostname parameter[3]. The thread spawns two concurrent co-routines via asyncio.gather(): recv_task and send_task[3].
4. **Inbound Data Ingestion (recv_task):** The read co-routine awaits incoming WebSocket frames using ws.recv() wrapped in a 0.5-second timeout[3]. Received payloads are capped at MAX_PAYLOAD_LEN characters and placed into the bounded inbound queue using non-blocking put_nowait() calls[3].
5. **Outbound Data Dispatch (send_task):** The write co-routine checks the outbound queue using non-blocking get_nowait() operations. When a broadcast payload is present, it is transmitted over the wire using await ws.send()[3].
6. **UI Polling & Decoupled Rendering:** The foreground UI layer uses isolated component fragments (@st.fragment(run_every="1s")) to poll the inbound queue every second without forcing a full script rerun. Polled messages are placed in a circular log buffer capped at MAX_BUFFER_LOGS items and rendered to the screen[3].

3.3 Structural Architecture Comparison Matrix

The table below contrasts standard client WebSocket implementations operating under RFC 6455 with the SAFD framework architecture:

Architectural Vector	Standard RFC 6455 Client Implementations	Proposed SAFD Framework Architecture
DNS Resolution Strategy	Dynamic resolution at connection time via OS resolver; vulnerable to TOCTOU DNS Rebinding[3].	Synchronously pre-resolved and pinned to IP literals in a bounded thread pool; Host and SNI headers mapped explicitly[3].
IPv6 Transition Defense	Shallow string or prefix matching; completely blind to 6to4, NAT64, Teredo, and ISATAP tunnels[5].	Recursive binary unwrapping of subterranean IPv6 transition payloads before boundary enforcement[5].
Concurrency & Threading Context	Runs directly on the main application thread or global event loop; locks script execution[1].	Decoupled background OS thread running an isolated asyncio loop[10].
Egress Backpressure Control	Unbounded OS socket buffering; vulnerable to memory exhaustion under high load[3].	Bounded queue.Queue primitives enforcing non-blocking put_nowait truncation[3].
UI Rendering Impact	Socket events force continuous script reruns or complete user interface lockup[1].	Localized polling fragments (@st.fragment) update log displays independently[1].
Lifecycle Teardown	Implicit context exits; vulnerable to orphaned background threads during hot-reloads[3].	Deterministic teardown using threading.Event, queue flushing, and explicit thread joining[10].

4. Technical Design Specification: Parameters & Task Management

4.1 System Operational Parameters

The SAFD architecture relies on five strict operational bounds to ensure deterministic resource consumption and performance:

Parameter Flag	Type	Default Value	Engineering Description & Constraint Bounds
MAX_PAYLOAD_LEN	int	2000	Maximum character length per frame buffer; truncates oversized messages to protect memory[3].
MAX_QUEUE_SIZE	int	100	Fixed capacity cap for inbound and outbound queues; engages backpressure when full.
MAX_BUFFER_LOGS	int	15	Size limit for the foreground circular display buffer; prevents DOM bloating during long sessions.
DNS_TIMEOUT	float	3.0s	Hard timeout limit for synchronous DNS resolution threads during

			ingress checks.
JOIN_TIMEOUT	float	3.5s	Boundary timeout for joining worker threads during connection teardown.

4.2 Full-Duplex Co-routine Execution Mechanics

Concurrent full-duplex communication inside the background worker thread is managed by asyncio.gather(), running recv_task and send_task simultaneously[3].

The inbound co-routine (recv_task) operates on an active loop conditioned on stop_event.is_set(). To ensure the co-routine routinely checks the shutdown signal without blocking indefinitely on idle sockets, the read call is wrapped in asyncio.wait_for(ws.recv(), timeout=0.5)[3]. If a timeout occurs, control yields back to the loop iteration. When a frame arrives, its length is checked against MAX_PAYLOAD_LEN. If the payload exceeds 2,000 characters, it is truncated before being enqueued via msg_queue.put_nowait()[3]. If the inbound queue is full due to backpressure, the frame is dropped safely without raising an unhandled exception[3].

The outbound co-routine (send_task) checks the egress queue on every iteration using send_queue.get_nowait(). If a payload string is retrieved, it is transmitted over the wire using await ws.send(payload)[3]. An outbound event log entry is then formatted and enqueued into msg_queue for UI display. If send_queue.get_nowait() raises a queue.Empty exception, the task yields execution to the event loop by calling await asyncio.sleep(0.1)[3].

4.3 Deterministic State Teardown Protocol

Disconnecting a stream or changing target endpoints requires a deterministic teardown sequence to prevent orphaned background threads, memory leaks, and socket exhaustion[10].

The disconnect_stream() procedure executes four sequential operations:

First, if an active shutdown event (ws_stop_event) exists in session state, the interface calls .set()[10]. This breaks the polling loops inside both recv_task and send_task.

Second, if a worker thread reference (ws_thread) exists, the main thread calls st.session_state.ws_thread.join(timeout=3.5)[10]. The JOIN_TIMEOUT bound ensures that the foreground thread will not hang indefinitely if an underlying TCP socket is stuck in a CLOSE_WAIT state[3].

Third, the worker thread's event loop completes pending tasks, exits loop.run_until_complete(), and releases resources via loop.close().

Fourth, the application state explicitly sets ws_thread, ws_stop_event, ws_queue, and ws_send_queue to None. This purges stale queue references and readies the application state for a clean re-initialization.

5. Strategic Architectural Insights & Industry Implications

Evaluating the SAFD framework reveals critical second- and third-order insights for security engineering and infrastructure design:

- **Egress Microservice Security:** Enforcing pre-resolved IP pinning at the application layer provides true defense-in-depth[3]. Traditional perimeter security architectures often rely on egress firewalls or proxy filters that evaluate initial hostnames[5]. By resolving target endpoints synchronously, unwrapping subterranean IPv6 transition headers, and binding sockets directly to verified IP literals, SAFD prevents SSRF pivots even when application containers are deployed in permissive cloud network segments[4].
- **Preserving Application Loop Stability:** Offloading networking loops to dedicated OS worker threads prevents network latency from affecting UI rendering[10]. Decoupling thread execution ensures that UI rendering metrics stay consistent regardless of network stability, payload throughput, or socket state transitions[1].
- **Predictable Memory Footprint:** Combining bounded inter-thread queues with hard payload character limits protects memory stability[3]. Under heavy traffic spikes, excess inbound frames are dropped at the queue boundary rather than filling memory buffers[3]. This design ensures long-term operational stability for web applications deployed on memory-constrained infrastructure.
- **Granular Component-Level Updates:** Using localized polling fragments (@st.fragment) eliminates the CPU overhead associated with full page script re-executions[1]. The UI thread updates only the specific log component tree, keeping CPU usage low during continuous full-duplex communication[1].

6. Architectural Conclusions

The Secure Asynchronous Full-Duplex Stream Framework (SAFD-FRAMEWORK-SPEC-01) delivers a definitive standard for hosting full-duplex WebSocket streams in script-reexecution application runtimes[1]. By combining strict ingress IP pinning, bitwise unwrapping of subterranean IPv6 transition mechanisms, thread-isolated asynchronous loops, and non-blocking UI fragment rendering, the architecture resolves the trade-offs between network flexibility, operational performance, and perimeter security[1]. Adopting this specification eliminates TOCTOU DNS Rebinding vulnerabilities, prevents subterranean SSRF exploits, and guarantees a responsive, memory-stable user experience across enterprise streaming applications[1].

Works cited

1. Using WebSockets (RFC 6455) | InterSystems IRIS Data Platform, https://docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls?KEY=AWEBSOCKETS

2. WebSocket Protocol: RFC 6455 Handshake, Frames & More, https://websocket.org/guides/websocket-protocol/
3. Client (threading) - websockets 17.0 documentation - Read the Docs, https://websockets.readthedocs.io/en/stable/reference/sync/client.html
4. RFC 6455 - The WebSocket Protocol - IETF Datatracker, https://datatracker.ietf.org/doc/html/rfc6455
5. IPv6 transition address bypass of SSRF guard in URL validation, https://github.com/microsoft/UFO/security/advisories/GHSA-7hrg-r8xr-p8gr
6. RFC 3964 - Security Considerations for 6to4 - IETF Datatracker, https://datatracker.ietf.org/doc/html/rfc3964
7. ISATAP - Wikipedia, https://en.wikipedia.org/wiki/ISATAP
8. How ISATAP Works and How It Can Help You Migrate to IPv6 - ARIN, https://www.arin.net/vault/blog/2013/05/14/how-isatap-works-and-how-it-can-help-you-migrate-to-ipv6-2/
9. Client (asyncio) - websockets 17.0 documentation - Read the Docs, https://websockets.readthedocs.io/en/stable/reference/asyncio/client.html
10. Client - websockets 17.0 documentation - Read the Docs, https://websockets.readthedocs.io/en/stable/faq/client.html
11. How Teredo Tunneling Provides IPv6 Connectivity Through NAT, https://oneuptime.com/blog/post/2026-03-20-teredo-tunneling-ipv6-nat/view
12. What is WebSocket? (RFC 6455) - 0AndWild_log, https://0andwild.com/en/posts/260210_websocket/
13. WebSocket - Wikipedia, https://en.wikipedia.org/wiki/WebSocket
14. IPv6 Transition Techniques - SharkFest, https://sharkfest.wireshark.org/retrospective/sfus/presentations12/BI-13_IPv6_Transition_Techniques.pdf
15. How to Understand ISATAP - OneUptime, https://oneuptime.com/blog/post/2026-03-20-understand-isatap/view
16. RFC 5214 - Intra-Site Automatic Tunnel Addressing Protocol (ISATAP), https://datatracker.ietf.org/doc/html/rfc5214
17. How to Configure ISATAP Tunneling for IPv6 on IPv4 Intranets, https://oneuptime.com/blog/post/2026-03-20-isatap-tunneling-ipv6-intranet/view
18. ISATAP Tunnel Support for IPv6 - Cisco, https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/interface/configuration/xe-16-12/ir-xe-16-12-book/ip6-isatap-xe.pdf/1000
19. K000132320: Overview of the Websocket profile - MyF5 | Support, https://my.f5.com/manage/s/article/K000132320
20. How to Connect a WebSocket Client to an IPv4 Server Address, https://oneuptime.com/blog/post/2026-03-20-websocket-client-ipv4/view