Expanding Multilingual Interoperability in the COGNIT Device Runtime

Integrating Additional Programming Languages into the Cognitive Cloud-Edge Continuum

Version 1.0
May 2026

Abstract

COGNIT is an AI-enabled Adaptive Serverless Framework for the Cognitive Cloud-Edge Continuum that enables the seamless, transparent and trustworthy integration of data processing resources across public providers and on-premises data centres. The Device Runtime is the on-device library that abstracts authentication, scheduling, Edge Cluster discovery, latency measurement and function offloading. It is currently distributed as two reference implementations: device-runtime-py (Python, v4.0, Nov 2025) and device-runtime-c (C, v4.0, Dec 2025), both released under Apache 2.0.

This white paper proposes the next evolutionary step of the framework: opening the Device Runtime to additional programming languages such as Rust, Go, Java/Kotlin, JavaScript/TypeScript and C++. Building directly on the source code and public wikis of both reference implementations, the document explains the real architecture currently in production, identifies the language-agnostic boundaries (REST endpoints, JSON envelope, Biscuit-token authentication, 16 KB function payload limit), and outlines integration approaches, architecture considerations, security and footprint constraints, and a set of best practices for delivering production-grade multi-language interoperability.

This project is funded by the European Union's Horizon Europe research and innovation programme under Grant Agreement 101092711 – SovereignEdge.Cognit

Document Metadata

Project Title:	A Cognitive Serverless Framework for the Cloud-Edge Continuum
Project Acronym:	SovereignEdge.COGNIT
Call:	HORIZON-CL4-2022-DATA-01-02
Grant Agreement:	101092711
Related Deliverable:	D3.5 — COGNIT FaaS Model — Scientific Report
Document Type:	White Paper / Technical Position Paper
Authors:	Álvaro Puente (Ikerlan) Ivan Valdés (Ikerlan) Aitor Garciandia (Ikerlan) Mikel Irazola (Ikerlan)
Reviewers:	Antonio Álvarez (OpenNebula Systems) Marco Mancini (OpenNebula Systems) Alberto P. Martí (OpenNebula Systems)
Dissemination Level:	PU: Public
Version:	1.0
Issue Date:	May 2026
Status:	Published

Executive Summary

The COGNIT Device Runtime is the on-device entry point through which any device, from a microcontroller-class sensor to a multi-core gateway or industrial PLC, interacts with a COGNIT deployment. The runtime hides authentication, requirement scheduling, Edge Cluster discovery, latency measurement and function offloading behind a small, opinionated API.

Today, the Device Runtime is shipped as two reference implementations: device-runtime-py (Python) and device-runtime-c (C). Both speak the same wire format (a JSON envelope over HTTPS, against five canonical REST endpoints exposed by the COGNIT Frontend and one endpoint exposed by the Edge Cluster Frontend) and both run the same four-state lifecycle. They differ, however, in how the user function is serialised: the Python client uses cloudpickle plus base64; the C client uses Protocol Buffers (nanopb) plus base64. Both encodings ride the same FC field of the same JSON message, the wire is uniform, the producer is per-language.

That asymmetry is the central insight of this white paper. The COGNIT framework is already prepared for multi-language expansion at the wire and protocol layers, but the function-encoding step is implementation-specific and must be replicated for every new language. The C client further demonstrates an explicit “portability via callbacks” pattern: HTTP, base64 and SHA-256 are not built into the library, they are delegated to integrator-supplied callbacks (the example uses libcurl and MbedTLS). This callback discipline is itself a valuable model for new bindings.

The document proposes (i) a canonical, frozen specification of the wire and protocol layers (cognit-spec); (ii) three integration approaches (native, FFI/binding over a C ABI, sidecar) plus a thin REST-wrapper variant (A.1) for any language, with explicit guidance on when each is acceptable; (iii) a prioritised roadmap that starts with Rust (mirrors the C client) and Go (mirrors the cloud-edge gateway use case); (iv) a set of best practices for cross-language conformance, observability and CI/CD; and (v) the matching evolution required in the Serverless Runtime (pluggable executors, artifact-oriented payloads, per-language sandboxing, flavour-driven scheduling and cold-start caching) so that the offloaded code can actually run in the target language, not be transparently translated back to Python on arrival. The expected outcome is a Device Runtime that behaves identically, and verifiably, regardless of the language in which the user writes the Device Client (Approaches A, B and C; the thin REST-wrapper variant A.1 is intentionally a lower-guarantee escape hatch and is not covered by this behavioural equivalence), while preserving the footprint, security and adaptability properties that define COGNIT.

Table of Contents

Abbreviations and Acronyms

ABI	Application Binary Interface
AI	Artificial Intelligence
API	Application Programming Interface
CFC	COGNIT Frontend Client (in-library)
CFE	COGNIT Frontend Engine (server-side)
CC	Confidential Computing
CI/CD	Continuous Integration / Continuous Delivery
DaaS	Data as a Service
ECF	Edge Cluster Frontend
ECFE	Edge Cluster Frontend Engine
FaaS	Function as a Service
FFI	Foreign Function Interface
HTTP/HTTPS	Hypertext Transfer Protocol (Secure)
IoT	Internet of Things
JNI	Java Native Interface
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MbedTLS	Mbed TLS cryptography library
MCU	Microcontroller Unit
mTLS	Mutual TLS
nanopb	Embedded-friendly C implementation of Protocol Buffers
Protobuf	Protocol Buffers
REST	Representational State Transfer
RTT	Round-Trip Time
SDK	Software Development Kit
SHA-256	Secure Hash Algorithm 256-bit
SR	Software Requirement
TLS	Transport Layer Security
WP	Work Package
WASM	WebAssembly

1. Introduction

COGNIT is a serverless framework that distributes the execution of functions across the cloud-edge continuum, hiding the underlying complexity from the developer. As described in deliverable D3.5, the framework is built around four essential components — Device Client, COGNIT Frontend, Edge Cluster and Serverless Runtimes — that cooperate to authenticate users, schedule workloads on the most suitable resources and execute functions in isolated environments.

The Device Client is the only COGNIT component that lives on the user's device. Its purpose is twofold: to expose a developer-friendly interface for offloading functions and to handle, transparently, every aspect of the communication with the rest of the framework. The Device Client is implemented as the Device Runtime library, currently distributed in Python (device-runtime-py) and C (device-runtime-c).

This white paper builds on the architectural foundations described in D3.5 and on the actual source code of both reference implementations (v4.0 in both cases) and proposes a structured path to integrate additional programming languages into the Device Runtime. The objective is not to replace the existing implementations, but to formalise a multi-language strategy that preserves a single, consistent behaviour across all language bindings.

1.1. Scope and Audience

This document targets three audiences: COGNIT core developers and architects who must decide which languages to support next, third-party contributors who may wish to develop and maintain a Device Runtime binding for a language not covered by the core team, and end users and integrators who need to understand the trade-offs of each binding to choose the most appropriate one for their hardware, workload and operational constraints.

1.2. Relation to D3.5 and to the Reference Repositories

This document targets three audiences: COGNIT core developers and architects who must decide which languages to support next, third-party contributors who may wish to develop and maintain a Device Runtime binding for a language not covered by the core team, and end users and integrators who need to understand the trade-offs of each binding to choose the most appropriate one for their hardware, workload and operational constraints.

- **device-runtime-py:** https://github.com/SovereignEdgeEU-COGNIT/device-runtime-py
- **device-runtime-c:** https://github.com/SovereignEdgeEU-COGNIT/device-runtime-c

Every claim in Sections 3, 4 and 7.6 of this document is anchored to the public API reference, models and source files of these two repositories and, for Section 7.6, of the companion serverless-runtime repository listed in Appendix A.

1.3. Document Structure

This document targets three audiences: COGNIT core developers and architects who must decide which languages to support next, third-party contributors who may wish to develop and maintain a Device Runtime binding for a language not covered by the core team, and end users and integrators who need to understand the trade-offs of each binding to choose the most appropriate one for their hardware, workload and operational constraints.

- **Section 2** presents the strategic rationale for adding more languages.
- **Section 3** summarises the real architecture of the current Python and C clients, including endpoints, models and state machine.
- **Section 4** describes the integration approaches (native, the thin-REST-wrapper variant A.1, FFI/binding, sidecar) and analyses their trade-offs.
- **Section 5** proposes a reference multi-language architecture and a portable State Machine specification.
- **Section 6** reviews the candidate languages and provides a prioritisation rationale.
- **Section 7** addresses cross-cutting concerns: payload size, security, latency telemetry, observability, concurrency and threading, and the corresponding changes required in the Serverless Runtime to execute functions in languages other than Python.
- **Section 8** compiles best practices for cross-language conformance, testing and CI/CD.
- **Section 9** surveys related multi-language SDKs from which the COGNIT effort can learn.
- **Section 10** concludes with a phased roadmap and recommended next steps.

2. Strategic Rationale

Adding more languages to the COGNIT Device Runtime is not an end in itself — it is a means to remove friction from adoption and to align the framework with the way real-world cloud-edge systems are built today. This section frames the four main drivers that justify the investment.

2.1. Reaching Developer Communities Where They Already Are

Python and C cover the two extremes of the developer spectrum: rapid prototyping and AI/ML on one end, embedded and resource-constrained programming on the other. However, a large fraction of cloud-edge workloads is written today in languages that fall in between: Rust (industrial gateways, memory-safe systems), Go (cloud-edge gateways and orchestration), Java/Kotlin (industrial automation, Android), JavaScript/TypeScript (Node.js and dashboards) and C++ (Linux-based industrial PCs). Forcing every developer in these communities to either write their device-side glue code in C or to ship a Python interpreter is a real adoption cost.

2.2. Matching Workloads to the Right Execution Environment

A polyglot Device Runtime improves the granularity with which workloads can be mapped to the runtime that best fits them. For example, a Rust client can stream sensor data with deterministic latency, a Go client can drive an HTTP/gRPC orchestration layer, and a Java client can integrate with a corporate identity provider — all of them offloading the actual computation to the same Edge Cluster, possibly to a Python-based Serverless Runtime.

2.3. Reducing Integration Complexity in Brownfield Deployments

Most real-world edge deployments are brownfield: they integrate with legacy industrial protocols, vendor SDKs, and existing applications written in a specific language. Providing a Device Runtime in the language already used by the host application avoids the introduction of an additional runtime, simplifies dependency management, and reduces operational risk.

2.4. Preparing the Ground for the Cognitive Continuum

COGNIT is designed as an adaptive, AI-driven framework. As the orchestrator becomes more sophisticated, multi-language support enlarges the placement search space (more device classes can become first-class participants) and provides the orchestrator with a more representative view of the continuum, which feeds directly into better adaptive decisions.

3. Architectural Baseline (as implemented today)

Before discussing how to integrate new languages, this section consolidates the architectural baseline of the existing Python and C clients, drawing exclusively on the public source repositories. It identifies, for each layer, what is language-agnostic and must be preserved by every new binding, and what is language-specific and can be re-implemented idiomatically.

3.1. Components of the Device Runtime

Figure 3.1 summarises the architecture of the current Device Runtime, splitting it into the components that live on the device and the COGNIT-side components they communicate with. The orange edges highlight the language-agnostic contracts; the orange box at the bottom of the device side is the only component whose internals diverge between Python and C.

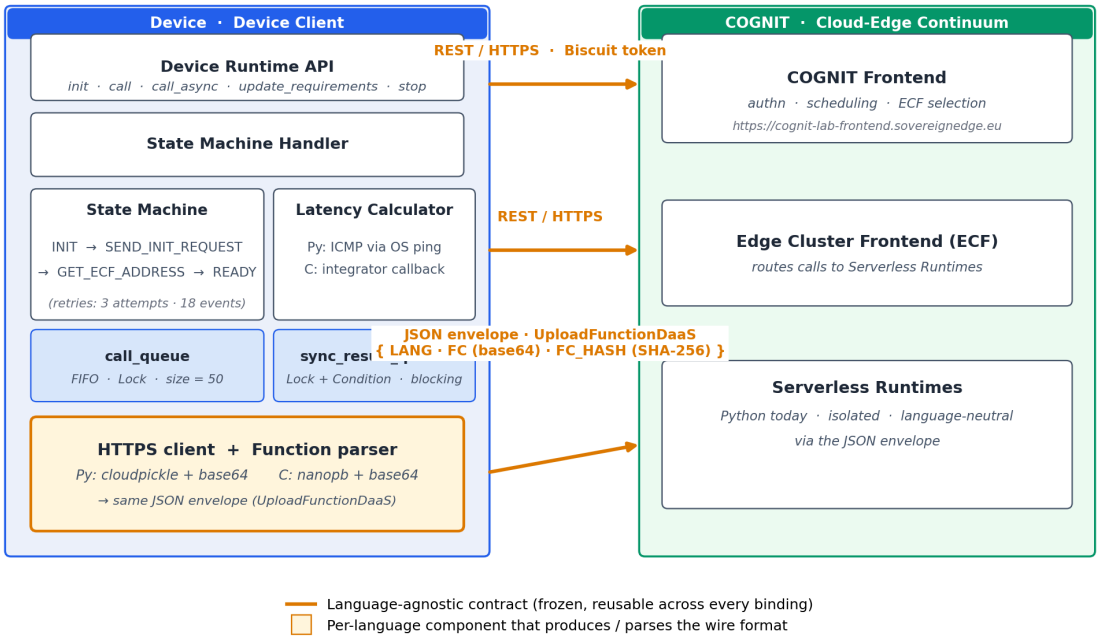

Figure 3.1. Device Runtime architecture — language-agnostic contracts highlighted.

Both implementations expose the same logical components — Device Runtime API, State Machine Handler, State Machine, Latency Calculator, call queue, sync result queue, HTTPS client and a function parser — but they realise them with very different abstractions. The Python implementation runs the State Machine as a background thread driven by StateMachineHandler.run(interval=0.05) (50 ms tick by default). The C implementation does not own the thread of control: device_runtime_call() is a synchronous, blocking call that drives state transitions inline. There is no asynchronous variant in the C client.

3.2. The Real Wire Contract

The wire contract is what every language binding must speak verbatim. It is built out of three building blocks observable in the source code of both clients.

3.2.1. REST endpoints exposed by the COGNIT Frontend and ECF

The C client declares the endpoint constants explicitly in cognit/include/cognit_frontend_cli.h and uses them throughout cognit/src/cognit_frontend_cli.c. The same endpoints are consumed by the Python client through the cognit.modules._cognit_frontend_client module. They are:

Verb	Endpoint (relative to cognit_frontend_endpoint)	Purpose
POST	`/v1/authenticate`	`Basic auth (username:password) → Biscuit token`
POST	`/v1/app_requirements`	`Upload Scheduling JSON; expects HTTP 200/201; body contains numeric app_req_id`
GET	`/v1/app_requirements/<app_req_id>/ec_fe`	`Resolve the optimal Edge Cluster Frontend; returns ecf_response_t metadata`
DELETE	`/v1/app_requirements/<app_req_id>`	`Release the application requirements (HTTP 204 expected)`
POST	`/v1/daas/upload`	`Upload a function (UploadFunctionDaaS); response body is a numeric fc_id`
POST	`<ECF endpoint>/{FAAS_REQUEST_ENDPOINT}`	`Synchronous FaaS execution against the resolved Edge Cluster Frontend`

Table 3.1. Real REST endpoints used by both Python and C clients (v4.0).

3.2.2. Scheduling model and Edge Cluster Frontend response

The Scheduling model is the JSON body of POST /v1/app_requirements. The Models and Enums wiki page of device-runtime-py defines the canonical field set; the C client declares a structurally equivalent struct in cognit_frontend_cli.h but, at v4.0, emits four of those fields under different JSON keys (see note below the table). The relevant fields, as named by the Python wiki, are:

Field	Type	Notes
FLAVOUR	`str`	`Mandatory. Default flavour used in the C example: "FaaS_generic_V2"`
ID	`str`	`Device identifier; used in DB lookups for caching`
GEOLOCATION	`Geolocation`	`{ latitude: float, longitude: float }`
MAX_LATENCY	`int (ms)`	`Optional. Maximum tolerated latency to the ECF in milliseconds`
MAX_FUNCTION_EXECUTION_TIME	`float (s)`	`Optional. Soft cap on per-function execution time`
MIN_ENERGY_RENEWABLE_USAGE	`int (%)`	`Optional. Minimum renewable-energy ratio (0–100)`

Table 3.2. Scheduling fields actually accepted by /v1/app_requirements (v4.0).

Note on field-name divergence between Python and C (v4.0). A direct comparison of the Python Scheduling pydantic model (cognit/models/_cognit_frontend_client.py) and the JSON actually emitted by the C client (cognit/src/cf_parser.c, cfparser_parse_app_requirements_as_json) shows that four of the eight fields use different JSON keys in the two implementations: ID (Python) vs. DEVICE_ID (C); MIN_ENERGY_RENEWABLE_USAGE (Python) vs. MIN_RENEWABLE_USAGE (C); PROVIDERS as list[str] (Python) vs. a single PROVIDER string (C); and IS_CONFIDENTIAL (Python) vs. CONFIDENTIAL_COMPUTING (C). The remaining four fields (FLAVOUR, MAX_LATENCY, MAX_FUNCTION_EXECUTION_TIME, GEOLOCATION) are emitted with identical keys by both clients. Reconciling this divergence — by picking one canonical naming and updating both clients accordingly — is one of the most concrete deliverables of the cognit-spec effort proposed in Section 5.2; until that happens, any new binding must choose which

dialect to follow (the Python wiki names are the more descriptive and are the ones used throughout this paper).

The response of GET /v1/app_requirements/<id>/ec_fe is the EdgeClusterFrontendResponse: { ID, NAME, HOSTS, DATASTORES, VNETS, TEMPLATE }. Both clients consume this response: the C client stores the full struct as ecf_resp inside cognit_frontend_cli_t, whereas the Python client extracts the cluster addresses from the TEMPLATE.EDGE_CLUSTER_FRONTEND field and keeps them in self.available_ecfs (a list of strings) inside CognitFrontendClient. The two clients therefore agree on the wire response but retain different shapes of it; the resolved address is used for every offloading call.

3.2.3. Function payload — UploadFunctionDaaS (cloudpickle in Python, nanopb in C)

Section 2.4 of D3.5 introduced the idea of a Protobuf wire format. The actual repositories show a more nuanced picture, illustrated in Figure 3.2: the JSON envelope and endpoints are shared, but the function-bytes producer differs between Python and C.

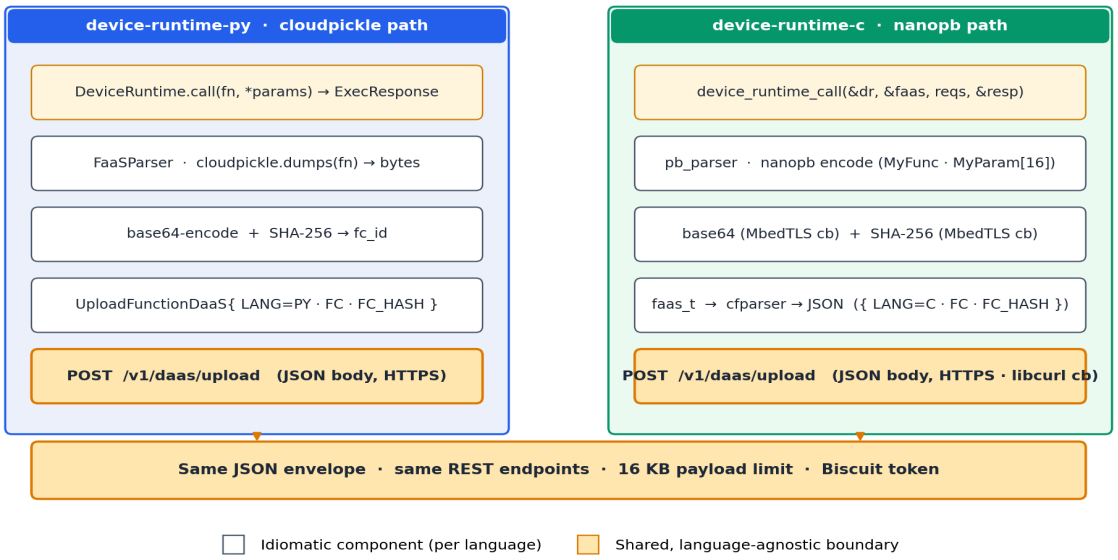

Figure 3.2. Two implementations, one wire format — Python and C client paths.

Both paths converge on the same JSON message UploadFunctionDaaS = { LANG, FC, FC_HASH }, where LANG is "PY" or "C", FC is the base64-encoded serialised function and FC_HASH is its SHA-256 (used as the function ID). The Python client produces FC by cloudpickle.dumps(fn) followed by base64; the C client produces FC by encoding faas_t with nanopb (a lightweight C implementation of Protocol Buffers, with MAX_PARAMS = 16) followed by base64. The maximum payload size is 16 384 bytes, defined by FAAS_MAX_SEND_PAYLOD_SIZE in device_runtime.h. Any new binding must therefore implement (a) a serialiser that preserves the ability to round-trip user functions in the target Serverless Runtime, and (b) the same JSON envelope. It should be noted, however, that today the Serverless Runtime ultimately interprets both paths as Python: the C path

transports the function body as a Python source string inside the nanopb MyFunc.fc_code field and executes it through Python's exec(). Section 7.6 analyses this asymmetry and the changes required on the server side for true multilingual offloading.

3.3. The Real State Machine

The State Machine has four canonical states in both implementations: INIT, SEND_INIT_REQUEST, GET_ECF_ADDRESS and READY. The C header device_runtime_state_machine.h declares them in the State_t enum, and the Python class DeviceRuntimeStateMachine declares the same four states. Figure 5.2 (Section 5.3) shows the canonical transitions.

Two retry counters bound the lifecycle: MAX_REQ_UPLOAD_ATTEMPTS = 3 for the upload of scheduling requirements, and MAX_GET_ADDRESS_ATTEMPTS = 3 for the resolution of the Edge Cluster Frontend address. When either limit is reached, the State Machine transitions back to INIT (LIMIT_REQUIREMENTS_UPLOAD or LIMIT_GET_ADDRESS).

The READY state is entered after ADDRESS_OBTAINED and is left only by (i) a token-invalidation event (TOKEN_NOT_VALID_READY) that loops back to INIT, (ii) a UPDATE_ECF_ADDRESS event that re-resolves the cluster on every offload, (iii) a READY_UPDATE_REQUIREMENTS event when the user calls update_requirements()/dr_sm_update_requirements(), or (iv) the explicit shutdown via stop()/device_runtime_free(), which DELETEs the application requirements on the COGNIT Frontend.

3.4. Public API Surface (Python and C side-by-side)

The user-facing API of the two implementations is small enough to reproduce verbatim. New bindings should remain at this level of granularity — anything more is a leaky abstraction; anything less leaves the user without async or update-requirements support.

Operation	Python (cognit.device_runtime)	C (device_runtime.h)
Construction	`DeviceRuntime(config_path: str)`	`cognit_config_t {endpoint, usr, pwd}`
Initialisation	`init(init_reqs: dict) -> bool`	`device_runtime_init(&dr, cfg, reqs, &faas)`
Update reqs.	`update_requirements(new_reqs: dict) -> bool`	`dr_sm_update_requirements(&sm, reqs)`
Sync call	`call(fn: Callable, *params) -> ExecResponse`	`device_runtime_call(&dr, &faas, reqs, &resp)`
Async call	`call_async(fn, callback, *params) -> bool`	— (not supported in v4.0 C client)

Shutdown	`stop() -> bool`	`device_runtime_free(&dr)`

Table 3.3. Public API surface of the two reference implementations (v4.0).

3.5. Latency, Queues and Concurrency

The Latency Calculator measures Round-Trip Time to candidate Edge Cluster Frontends. The Python implementation parses the output of the OS-level ping command (LatencyCalculator.calculate(ip)); the C implementation delegates the measurement to the user via callbacks. The probe count and aggregation function are not normatively specified by the wikis, which is one of the items to formalise in cognit-spec (Section 5).

The Python client maintains a CallQueue (FIFO list[Call] guarded by a Lock, soft size limit of 50, non-blocking add) and a SyncResultQueue (single-slot, blocking via Lock + Condition) so that call() blocks the calling thread until the result arrives, while call_async() returns immediately and the result is delivered through the user-supplied callback. The C client has no explicit queues — it offloads inline, one function at a time, blocking the calling thread.

3.6. The Callback-Based Portability Pattern (C client)

The C library deliberately does not link against any HTTP, base64 or hash library. It exposes three function-pointer hooks that the integrator must implement:

```C/C++
int my_http_send_req_cb(const char* buf, size_t size, http_config_t*
cfg);
int cognit_base64_encode(unsigned char b64[], size_t cap, size_t*
out_len, char in[], int in_len);
int cognit_hash(const unsigned char* str, size_t str_len, unsigned char
hash[]);
```

The minimal example wires them to libcurl (HTTP) and MbedTLS (base64, SHA-256), but any equivalent stack works. This “portability via callbacks” pattern is itself a useful precedent: it shows that the C library has been designed to be ported across vendor SDKs and embedded environments without modifying the COGNIT logic. The same pattern can be reused as the C ABI for an FFI-style binding (Approach B in Section 4).

4. Integration Approaches

There are three structurally different ways to bring a new language into the Device Runtime, plus one degenerate variant of the first one (Approach A.1) that is included for completeness. They differ in implementation effort, footprint, and operational complexity.

4.1. Approach A — Native Implementation

A native implementation re-writes the State Machine, the HTTPS client and the function parser directly in the target language, using only that language's standard libraries and ecosystem. This is how the existing Python and C implementations are built.

Pros: minimal runtime footprint; idiomatic API; no FFI overhead; first-class tooling.

Cons: highest implementation effort; the State Machine logic must be maintained in N codebases; cross-language behavioural drift is a real risk.

When to use: for any language intended to run on extreme-edge devices, and for any language whose ecosystem rejects FFI dependencies (Rust, Go).

4.1.1. Approach A.1 — Thin REST Wrapper (no State Machine)

A useful degenerate case of Approach A is to expose the COGNIT wire contract directly, without re-implementing the runtime logic. Instead of porting the State Machine, the Latency Calculator, the call queue and the function parser, the binding is reduced to a thin wrapper around the REST endpoints of Table 3.1: an HTTP client (or even a handful of curl-style snippets) that authenticates against /v1/authenticate, posts the Scheduling JSON to /v1/app_requirements, resolves the Edge Cluster Frontend via /v1/app_requirements/<id>/ec_fe and finally posts the UploadFunctionDaaS envelope (LANG, FC, FC_HASH) to /v1/daas/upload and to the FaaS execution endpoint. The state, the retries, the token refresh logic and the cluster re-resolution are pushed back onto the user.

Pros: lowest possible implementation effort — any language with an HTTPS and JSON library can interact with COGNIT in a few hundred lines of code; zero per-language porting of the State Machine; ideal for scripts, notebooks, ad-hoc tools and bootstrapping a new ecosystem; doubles as a reference implementation against which the Tier 1 wire-conformance tests (Section 5.4) can be executed.

Cons: the user must manage authentication and Biscuit-token renewal manually, implement retries and back-off, decide when to re-resolve the Edge Cluster Frontend on failure, and handle their own latency probing if they wish to inform the orchestrator; there is no async path, no call queue, no built-in observability or structured logging; the binding does not deliver an idiomatic developer experience and offers significantly less control over correctness than Approach A. In short, you can talk to COGNIT from any language, but

you are no longer using a Device Runtime — you are using the COGNIT REST API directly, and the protective layer that the runtime normally provides is absent.

When to use: as a quick-start to validate end-to-end connectivity in a new language before committing to Approach A; as the reference client used by the conformance suite itself; for one-off integrations, scripting, classroom demonstrations or research notebooks where the engineering investment of a full native binding is not justified. It is not recommended for production deployments, for extreme-edge devices that require careful retry/backoff, or for any scenario where the orchestrator's adaptive decisions depend on consistent latency telemetry from the device.

4.2. Approach B — FFI / Binding over the Existing C Core

The C library is already a candidate "core": its public surface (device_runtime_init, device_runtime_call, device_runtime_free) plus the three callback hooks define a stable C ABI. Each new language binding becomes a thin layer that translates between the language's idioms and that ABI.

Pros: single source of truth for the protocol logic; new languages can be added with low effort once the binding generator is set up.

Cons: introduces a build-time dependency on a native library; debugging across the FFI boundary is harder; some ecosystems (browsers, restricted JVMs) are hostile to FFI; the integrator must still supply the HTTP/base64/hash callbacks.

When to use: for languages that have mature FFI support and whose users typically deploy on devices that can host a small native dependency, e.g. Java/Kotlin (JNI / FFM API), Python (cffi, PyO3) or Node.js (N-API).

4.3. Approach C — Sidecar / Local Agent

Here, the Device Runtime is implemented as a separate process (the "sidecar") that exposes a localhost gRPC or HTTP API. The user's application, in any language, talks to the sidecar; the sidecar talks to the COGNIT Frontend and the Edge Cluster Frontend.

Pros: zero per-language work — any language with an HTTP client can use COGNIT.

Cons: adds a process and a memory budget; introduces an extra IPC hop; complicates packaging; incompatible with extreme-edge nodes.

When to use: as a temporary bootstrap mechanism while a proper native or FFI binding is being developed. SR1.4 (low memory footprint) explicitly forbids this approach on extreme-edge nodes.

4.4. Comparative Summary

Criterion	A — Native	B — FFI over C	C — Sidecar
Footprint	Best (per-language)	Good (shared C core)	Worst (extra process)
Effort per new language	High	Medium	Very low
Behavioural consistency	Requires conformance suite	Strong (single core)	Strong (single core)
Async support	Whatever the language offers	Limited (C core is sync)	Limited (sync IPC hop)
Suitable for extreme-edge	Yes	Sometimes	No
Suitable for cloud-edge	Yes	Yes	Yes

Table 4.1. Trade-offs between integration approaches.

5. Reference Multi-Language Architecture

To prevent uncontrolled divergence between bindings, this white paper proposes a reference architecture organised in four layers (Figure 5.1) and supported by a central specification repository (cognit-spec).

5.1. Layered View

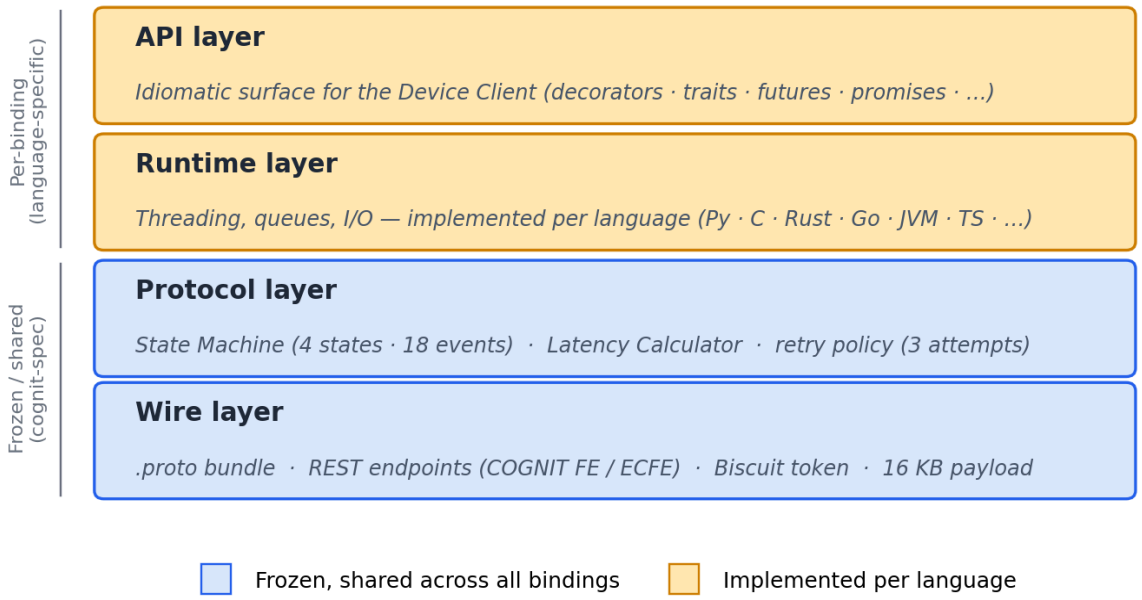

Figure 5.1. Layered reference architecture for the multi-language Device Runtime.

From bottom to top:

1. Wire layer — REST endpoints, JSON envelope (UploadFunctionDaaS, Scheduling, ECF response), Biscuit-token authentication, 16 KB payload limit.
2. Protocol layer — State Machine specification (4 states and the full set of named events listed in cognit-spec), Latency Calculator algorithm, retry/backoff policy (3 attempts).
3. Runtime layer — language-specific implementation of queues, threading, I/O.
4. API layer — idiomatic surface exposed to the Device Client (Python decorators, Rust traits, Java annotations, etc.).

The wire and protocol layers are versioned and frozen in cognit-spec. The runtime and API layers are owned by each binding.

5.2. The Canonical cognit-spec Repository

cognit-spec is proposed as a stand-alone repository that owns:

- OpenAPI 3.1 specification of every endpoint listed in Table 3.1, with example payloads and HTTP status codes (200/201 for success, 204 for DELETE, others as documented).
- JSON Schemas for Scheduling, UploadFunctionDaaS, ExecResponse and EdgeClusterFrontendResponse, derived directly from the Python wiki and the C structs.
- .proto bundle for the C/Rust path (MyFunc, MyParam, FaasResponse — to be re-extracted from cognit/include/pb_parser.h and the generated nanopb code), plus a versioning policy.
- State-transition table encoding the 4 states and the full set of named events (ADDRESS_OBTAINED, REQUIREMENTS_UP, SUCCESS_AUTH, etc.) with their guard conditions and the retry counters (3 attempts).
- Conformance test suite (see Section 5.4).

5.3. Portable State Machine

Figure 5.2 makes the four-state lifecycle explicit, including the events actually used by the existing implementations.

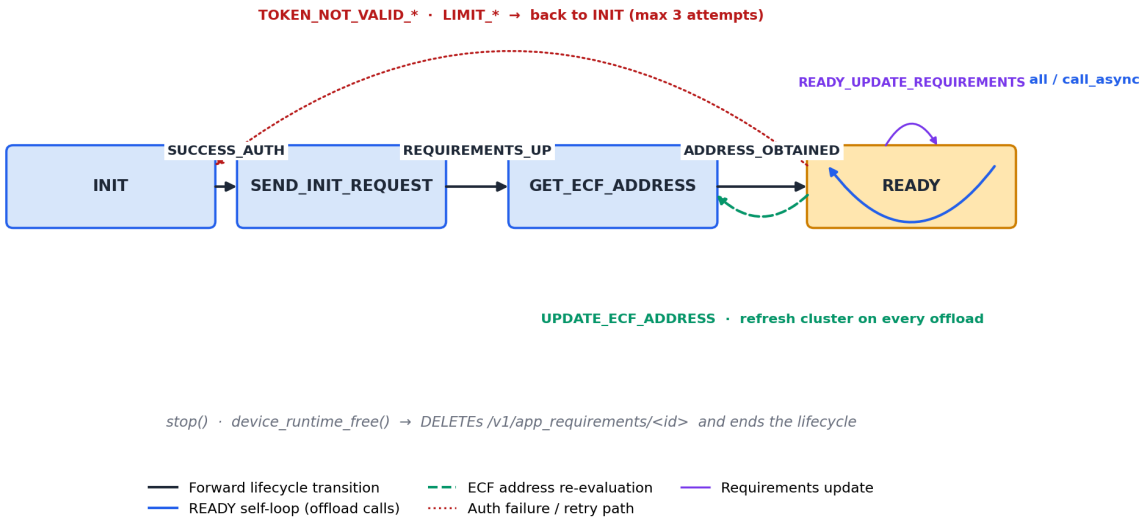

Figure 5.2. Device Runtime State Machine — 4 canonical states.

Notable behaviour preserved by both implementations: every offloading call from READY first triggers UPDATE_ECF_ADDRESS to refresh the cluster (so the orchestrator can react to changes), and an invalid token from any state returns control to INIT, where re-authentication is attempted up to MAX_REQ_UPLOAD_ATTEMPTS times before the lifecycle is reset.

5.4. Conformance Test Suite

A central conformance test suite is the only practical way to keep N native bindings aligned. It is organised in three tiers:

- **Tier 1** — Wire conformance: replay recorded REST exchanges (against a mock built from cognit-spec) and check that the binding produces byte-identical JSON requests and accepts the documented responses.

- **Tier 2** — Protocol conformance: drive the binding through every state and event of the State Machine, including failure injection (network drops, expired tokens, retry-limit exhaustion).

- **Tier 3** — End-to-end conformance: run a small reference workload (the my_calc example from D3.5 §2.5 or the minimal-offload-example shipped with device-runtime-c) against a real COGNIT instance and validate latency, payload size and correctness.

6. Candidate Languages and Prioritisation

Not every language deserves the same investment. The candidates below are ordered by the strength of their fit for the existing architecture.

6.1. Rust

Rust is the most natural next step. Its ownership model and zero-cost abstractions deliver C-grade footprints with much stronger safety guarantees, which fits SR1.4 (low memory footprint) and SR1.5 (security). The crate prost provides idiomatic Protobuf bindings that, once the .proto files have been re-extracted into cognit-spec (Section 5.2), can consume them unchanged, so the Rust binding can reuse the C client's function-encoding semantics directly. tokio gives a clean async story for the Device Runtime API. Rust also opens the door to first-class WebAssembly compilation for browser-based or sandboxed targets.

6.2. Go

Go is the de facto language for cloud-edge gateways and orchestration. The standard library covers HTTPS, JSON and goroutines out of the box; protoc-gen-go is the reference Protobuf code generator. A native Go binding fits the "gateway agent" deployment model very well and can also host Approach C (sidecar) if needed.

6.3. Java and Kotlin

Java and Kotlin remain dominant in industrial and enterprise environments, including Android-based industrial HMIs. Two implementation strategies are reasonable: a pure-JVM native implementation, or an FFI binding over the C core via the Foreign Function & Memory API (JEP 442). Kotlin Multiplatform additionally allows a single binding to target JVM, Android and Kotlin/Native.

6.4. JavaScript / TypeScript

TypeScript is the right choice when COGNIT must integrate with a Node.js back-end, a browser-based dashboard, or an Electron-based industrial console. Native HTTPS and Protobuf libraries (protobufjs, ts-proto) are mature. Browser deployments require careful handling of the Biscuit token (no localStorage on long-lived devices) and CORS.

6.5. C++

A native C++ implementation can coexist with the C one and offer a more idiomatic API to existing C++ codebases (RAII, smart pointers, std::function instead of the current callback structs). For severely constrained MCU targets, the C implementation should remain the recommended option.

6.6. Other Candidates

C# / .NET, Swift, Dart/Flutter and Lua are good candidates for an FFI or sidecar strategy once the native bindings above are stabilised.

6.7. Proposed Prioritisation

Wave	Language	Recommended approach	Rationale (anchored in the C/Python clients)
1	Rust	Native	prost can reuse the C client's .proto files once they are re-extracted into cognit-spec; tokio gives a clean async API; ideal for high-end edge nodes and WebAssembly.
1	Go	Native	Mature std-lib, protoc-gen-go, fits gateway agents and the optional sidecar deployment of Approach C.
2	Java / Kotlin	FFI over C	JNI / FFM API can wrap the existing C library; Kotlin Multiplatform reaches JVM, Android and Kotlin/Native.
2	TypeScript / Node.js	Native	protobufjs/ts-proto for the C-style payload; cloudpickle path is irrelevant because TS users do not author Python functions.
3	C++	Native (alongside C)	Idiomatic API for Linux-based industrial PCs; can wrap the C core internally.
3	C# / .NET, Swift, Dart	FFI or Sidecar	Vertical-specific reach; lower priority until the core matures.

Table 6.1. Proposed prioritisation of new Device Runtime bindings.

7. Cross-Cutting Concerns

7.1. Payload Limits and Memory Footprint (SR1.4)

The C client enforces a hard FaaS payload cap of 16384 bytes (FAAS_MAX_SEND_PAYLOD_SIZE), and nanopb is configured for at most MAX_PARAMS = 16 parameters per call. New bindings must respect both limits, even if the target language could in principle handle larger messages, because the receiving Serverless Runtime will not.

7.2. Security (SR1.5)

Each binding must enforce the same security baseline:

- Mandatory TLS for every call (the C constant STR_PROTOCOL is "https"; basic authentication is exchanged for a Biscuit token on the very first request).
- Biscuit token transported as a custom token: HTTP header on every subsequent request, as in the v4.0 reference clients. Migration to Authorization: Bearer is recommended once cognit-spec freezes this detail.
- Safe storage of credentials (no plaintext on disk; integration with platform-specific credential stores when available).
- Validation of every JSON response against the cognit-spec schemas before processing.
- Optional support for mTLS and Confidential Computing attestation tokens, aligned with SR6.2 and the IS_CONFIDENTIAL Scheduling field.

7.3. Latency Telemetry (SR1.6)

All bindings must include a Latency Calculator capable of measuring RTT to candidate Edge Cluster Frontends. The Python client uses ICMP via the OS ping command; the C client delegates to a user callback. cognit-spec should normatively define the probe count and aggregation function so that the orchestrator’s view of the continuum stays consistent across bindings.

7.4. Observability and Diagnostics

A multi-language deployment must be diagnosable as a single system. Bindings should expose:

- Structured logging with a common schema (timestamp, state, event, app_req_id, fc_id, correlation ID). The C client already has COGNIT_LOG_TRACE/DEBUG/INFO/ERROR macros that can serve as a model.

- Metrics following a common naming convention (e.g., cognit.device.calls.total, cognit.device.latency.ms, cognit.device.payload.bytes).

- Optional OpenTelemetry tracing across the Device Client, the COGNIT Frontend and the Edge Cluster Frontend.

7.5. Concurrency and Threading

Each binding chooses its own threading model (threads, coroutines, async/await, goroutines). Three invariants must hold regardless:

- A synchronous call() must remain blocking until a result or a timeout is produced (today, the C client default timeout is ECF_REQ_TIMEOUT and the Python timeout is configurable per call).

- An asynchronous variant (where the language offers one) must return immediately and deliver the result via callback, future, promise or channel.

- The State Machine must observe a single logical owner thread, to avoid race conditions during state transitions.

7.6. Serverless Runtime Implications for Multilingual Execution

Everything discussed up to this point in Sections 3 through 7.5 addresses the device side of the equation: how a Device Runtime, written in language X, can serialise a function and ride the wire contract to COGNIT. That is only half of the multilingual story. The other half is the receiving end — the Serverless Runtime (repository SovereignEdgeEU-COGNIT/serverless-runtime) — which is the service deployed on the scheduled Edge Cluster node and is responsible for actually executing the offloaded function. Inspection of the public source code (FastAPI application under app/, endpoint POST /v1/faas/execute-sync, model ExecSyncParams) shows that the Serverless Runtime, although it accepts a lang field that today can take the values "PY" or "C", ultimately converges on a single executor class, PyExec, in app/modules/_pyexec.py. The C path simply extracts the embedded source code (the fc_code field of the protobuf MyFunc message) and runs it through Python's built-in exec() (see make_fc_executable in app/api/v1/faas.py). In other words, today the C client is not a vehicle for offloading C code: it is a memory-efficient transport for offloading Python code from a constrained device. Every Device Runtime binding proposed in this paper, if naively added, would inherit the same constraint.

To make multilingual offloading truly end-to-end, the Serverless Runtime must therefore evolve in parallel with the Device Runtime. Five concrete changes are required, and they can be staged independently of the Device Runtime bindings of Sections 4 to 6.

- **Pluggable executor abstraction.** The current Executor base class in app/modules/_executor.py declares only run() and get_result() and is sub-classed

only by PyExec. A first step is to formalise this abstraction into a stable contract (lifecycle, error codes, status, telemetry hooks, execution time bounds) and to introduce sibling implementations such as WasmExec, JvmExec, NodeExec and NativeExec. The dispatch in execute_sync becomes a registry lookup keyed on the lang field of ExecSyncParams, replacing the hard-coded if/elif branches.

- **Artifact-oriented payload semantics.** The current UploadFunctionDaaS envelope assumes that FC is either a cloudpickled Python callable or a protobuf-encoded Python source string. Real multilingual offloading requires FC to be re-interpreted as a generic artifact whose meaning is governed by LANG: a WebAssembly module for WASM; class bytes or a small JAR for JVM; ESM source for JS/TS; a content-addressed reference (e.g. an OCI digest) to a pre-built native object for NATIVE. The 16 KB FAAS_MAX_SEND_PAYLOD_SIZE ceiling (Section 7.1) is incompatible with naive inlining of, say, a Rust-to-WASM binary; for these cases the payload should carry a reference resolvable by MinIO/RabbitMQ (both already wired into the Serverless Runtime via app/modules/_minio_client.py and _rabbitmq_client.py) so that large artifacts can be uploaded out-of-band. The corresponding parameter envelope (MyParam in nano_pb2) is already protobuf-based and can be reused unchanged across languages, which makes it the obvious canonical wire format for parameters and results in the cognit-spec proposed in Section 5.2.

- **Per-language sandboxing and isolation.** The current exec()-based execution model is a known security weakness that any multilingual evolution should not propagate. WebAssembly (via wasmtime or wasmer) is the strongest candidate for a default secondary executor: it offers per-call memory caps, capability-based system calls (WASI preview 2), deterministic compilation and a unified compilation target for Rust, Go (TinyGo), C, C++, AssemblyScript and even Python via Pyodide. For interpreted ecosystems, isolation can be achieved with subprocess + nsjail/bwrap or with the lightweight VM that already hosts the Serverless Runtime (the file /var/run/one-context/one_env referenced by get_vmid() shows that one VM is provisioned per Serverless Runtime instance by OpenNebula; per-language sandboxing can layer on top of this with minimal additional surface). Every new executor must enforce three invariants: CPU and wall-clock budget (today captured loosely by start_pyexec_time/end_pyexec_time), maximum resident memory and an explicit syscall/network policy. The Scheduling field MAX_FUNCTION_EXECUTION_TIME (Table 3.2) should be honoured by the sandbox, not merely observed by Prometheus after the fact.

- **Flavour-driven scheduling.** The Scheduling field FLAVOUR (today exclusively FaaS_generic_V2) becomes the lever by which the orchestrator routes a workload to a Serverless Runtime that supports the device's language. Each new executor type implies at least one new flavour (e.g. FaaS_wasm_V1, FaaS_jvm_V1, FaaS_node_V1), advertised by the Edge Cluster Frontend on startup. This in turn means that the Serverless Runtime VM image — built and provisioned via

system_unit/ and the uvicorn main:app --flavour startup contract — must be able to declare which executors and which language toolchains it bundles. Operators can then ship slim, single-language images for cost-sensitive deployments, or a fat "polyglot" image for development clusters.

- **Cold-start, caching and metrics.** Native, WASM and JVM executors all carry a non-trivial first-invocation cost (module compilation, class loading, JIT warm-up). The Serverless Runtime should maintain a per-flavour artifact cache keyed by FC_HASH (already present in UploadFunctionDaaS), pre-warm a small pool of runtimes per language, and emit cold-start metrics alongside the existing Prometheus histograms (sr_histogram_func_exec_time_seconds, function_duration_seconds) extended with a lang label so the orchestrator can compare cost across languages. This in turn closes the loop with Section 2.4: a richer cost model lets the adaptive placement engine make better polyglot decisions.

Notably, none of the five changes above invalidates the REST contract of Table 3.1 or the State Machine of Section 3.3: they live entirely behind the POST /v1/faas/execute-sync boundary. Multilingual Device Runtime bindings (Sections 4 to 6) and multilingual Serverless Runtime executors can therefore be evolved on independent schedules, with cognit-spec serving as the common contract between them. A pragmatic minimum-viable sequence is: (i) add a WasmExec behind a new FaaS_wasm_V1 flavour as the first non-Python executor; (ii) compile the Rust Device Runtime function (Section 6.1) to WASM end-to-end as the proof of concept; (iii) generalise the artifact format and the cache to cover JVM and Node next.

8. Best Practices

8.1. Single Source of Truth

cognit-spec owns the OpenAPI, the JSON Schemas, the .proto bundle, the State Machine specification and the conformance suite. Every binding imports the contract from this repository and never redefines it locally.

8.2. Idiomatic, Not Identical, APIs

Behaviour must be identical across bindings, but the API surface must feel native in each language. For example, call() may be a blocking function in Python, an async fn in Rust, a CompletableFuture in Java and a Promise in TypeScript, all sharing the same Scheduling parameter and ExecResponse return type.

8.3. Continuous Conformance

Every pull request to a binding must run Tier 1 and Tier 2 of the conformance suite in CI; Tier 3 is run on a schedule against a staging COGNIT environment.

8.4. Versioned Releases and Compatibility Matrix

Bindings must be released with explicit version numbers and documented compatibility against COGNIT Frontend versions. The current public reference is v4.0 for both Python and C.

8.5. Reference Sample Applications

Each binding should ship a small reference application that mirrors the existing examples (examples/cognit-template.yml + a my_calc style function in Python;
minimal-offload-example in C) so that newcomers can validate their environment in a few minutes.

8.6. Community Governance

Maintaining N bindings requires a clear governance model: a CODEOWNERS file per binding, a contribution guide, a security policy, and a release calendar that aligns binding releases with COGNIT Frontend releases.

9. Related Work

The COGNIT effort can borrow heavily from comparable multi-language SDKs in the cloud-native ecosystem:

- **gRPC SDKs** demonstrate how to keep more than ten language bindings behaviourally identical via a shared .proto registry and per-language code generators.
- **OpenTelemetry SDKs** (twelve languages) show the value of a strict specification repository (opentelemetry-specification) feeding a per-language conformance test suite.
- **Dapr** implements Approach C in production: a sidecar speaks gRPC and HTTP to user applications in any language, while the heavy lifting stays in a single Go binary.
- **AWS SDK** uses smithy as the canonical service definition language and generates idiomatic SDKs for nine languages from it.

The lessons distilled from these projects—formal specification first, conformance suite second, idiomatic APIs third—directly inform the cognit-spec design proposed in Section 5.

10. Roadmap and Conclusion

10.1. Phased Roadmap

- Phase 0 — Foundations. Extract the wire contract (Tables 3.1, 3.2 and the UploadFunctionDaaS schema) and the State Machine specification into cognit-spec. Bootstrap the conformance suite using device-runtime-py and device-runtime-c v4.0 as the first two compliant bindings.

- Phase 1 — Wave 1 bindings, plus first non-Python executor. Develop native Rust and Go bindings; reach Tier 3 conformance against a reference COGNIT instance. In parallel, land a WasmExec in the Serverless Runtime behind a new FaaS_wasm_V1 flavour (Section 7.6) and validate Rust → WASM end-to-end as the first true polyglot offloading path.

- Phase 2 — Wave 2 bindings and matching executors. Develop Java/Kotlin (FFI over the C core) and TypeScript (native) bindings; ship JvmExec and NodeExec in the Serverless Runtime so that Java and TypeScript workloads can actually run remotely, not merely talk to COGNIT; integrate the resulting end-to-end paths in industrial and enterprise pilots.

- Phase 3 — Wave 3 bindings, plus NativeExec. Add C++ and one FFI/sidecar-based language (e.g., C# or Swift) based on adoption signals; complete the Serverless Runtime executor matrix by shipping NativeExec (sandboxed native binaries) so that C, C++ and Rust workloads can execute directly without going through the WASM compilation step when latency or hardware-specific code paths demand it.

- Phase 4 — Continuous improvement. Track footprint, security and latency benchmarks across bindings; feed results into the orchestrator’s placement model.

10.2. Conclusion

Expanding the COGNIT Device Runtime beyond Python and C is a low-risk, high-impact step. The architectural foundations laid down in D3.5 and confirmed by the v4.0 source code, REST contracts, Biscuit-based authentication, a shared JSON envelope and a small, well-defined state machine, are already language-neutral. What is missing is a formalised reference architecture, a conformance test suite and a prioritised roadmap. This white paper provides all three. By following them, COGNIT can become a truly polyglot serverless framework for the cloud-edge continuum, capable of meeting developers and operators in the language and ecosystem they already use, without compromising on the footprint, security and adaptability properties that define the platform.