A Capability Broker for Workflow–Network QoS Coordination in B5G/6G Industrial Services

Qize Guo*, Yan Chen*, Tarik Taleb*, Bjoern Riemer†, Hemant Zope† and Hao Yu‡
*Faculty of Electrical Engineering and Information Technology, Ruhr University Bochum, Bochum, 44801, Germany
†Fraunhofer Institute for Open Communication Systems (FOKUS), Berlin, 10589, Germany
‡Hangzhou International Innovation Institute, Beihang University, Hangzhou, 31115, China
Email: {qize.guo, yan.chen-j3l, tarik.taleb}@rub.de, {bjoern.riemer, hemant.zope}@fokus.fraunhofer.de, haoyu1@buaa.edu.cn

***Abstract*—Industrial services in beyond-fifth-generation (B5G) and sixth-generation (6G) networks are increasingly executed as multi-phase workflows whose quality-of-service (QoS) demands change across ordered phases. Existing exposure, analytics, and policy-control functions can provide capability information and enforce QoS treatment, but they do not track the admitted QoS agreement between a workflow and the network. This paper studies this missing coordination function in the software control plane. We propose a Capability Broker that represents each admitted trajectory as a lifecycle-managed QoS commitment, including the workflow demand, network capability, validity time, and enforcement state. The Broker operates above existing exposure and policy-control interfaces and enforces capability freshness, duplicate-safe admission, legal state transitions, per-commitment event ordering, recovery routing, and Broker–network state consistency. Prototype results with microbenchmarks, fault injection, ablation, and controlled message loss show that the full Broker preserves commitment consistency with microsecond-level local processing overhead, while removing individual Broker functions exposes duplicate commitments, stale admissions, illegal transitions, lost-update divergences, or Broker–network inconsistency.**

***Index Terms*—B5G/6G, network softwarization, service orchestration, control-plane software, commitment lifecycle, QoS management**

I. Introduction

Industrial services in beyond-fifth-generation (B5G) and sixth-generation (6G) networks increasingly coordinate communication with the execution of an operational workflow, not just with an isolated traffic flow [1]–[3]. Consider a mobile inspection robot or production-cell controller that first exchanges low-rate telemetry and control commands, then uploads high-rate sensing data or video during inspection, and later switches to reporting or recovery after a defect or network degradation is detected [4], [5]. The application may choose among full-quality, reduced-quality, and fallback quality-of-service (QoS) modes for each phase. These choices are coupled: a reduced video mode can extend the inspection phase, lower the confidence of a downstream decision, or constrain which recovery mode remains feasible later. The communication problem is not simply to request one QoS treatment at a time. The control plane also needs to track the admitted QoS agreement as the workflow starts, adapts, recovers, and finishes.

Current 5G systems already provide building blocks. The Network Exposure Function (NEF) exposes selected capabilities and events, the Network Data Analytics Function (NWDAF) provides analytics about service experience and load, and the Policy Control Function (PCF) authorizes QoS treatment, including Alternative QoS Profile (AQP) fallback [6]–[9]. Their dominant interaction model remains request-driven and session scoped. They can provide capability information and enforce QoS treatment, but they do not define a runtime object that binds a multi-phase workflow demand to a network capability view.

This gap appears when workflow orchestration uses network capability information. Suppose the network provides a capability envelope that describes near-term supportability, and the application submits a demand trajectory that maps upcoming workflow phases to selected QoS modes and fallback options. Once admitted, this trajectory is no longer a single QoS request. It becomes a workflow–network QoS commitment: an admitted agreement that links workflow demand, network capability, validity time, and enforcement state.

Managing this commitment requires a software control-plane function. The Broker must check capability freshness before network-side policy actions, map retransmitted submissions to the same commitment, prevent workflow execution before network-side admission succeeds, route capability changes to recovery, and order asynchronous workflow-side and network-side events under the same commitment identity. Without such an owner, the control plane may use stale capability, create duplicate commitments, accept illegal workflow progress, or lose consistency with the network-side state.

Compared with the capability-aware workflow coordination vision in [10], this paper studies the runtime layer left open after admission: how an admitted trajectory is represented, checked, serialized, recovered, and synchronized. The novelty is not a new QoS application programming interface (API) or a new workflow scheduler. It is the commitment-management layer required after capability-aware admission has been made. The Broker operates between the Industrial Agent and the Network Agent, turns an admitted demand trajectory into a lifecycle-managed QoS commitment, and keeps this commitment consistent during admission, activation, recovery, release, and abort.

The main contributions are as follows:

- We formulate workflow–network QoS coordination as a software control-plane problem for capability-aware B5G/6G industrial services. The key abstraction is the capability commitment defined in Section II.
- We design a Capability Broker that turns exposed network capability from information into a lifecycle-managed service orchestration object. Its six software responsibilities are summarized in Table I.
- We implement a Broker prototype and evaluate it through local overhead measurement, fault injection, ablation, and lossy signaling experiments.

II. System Context and Commitment Model

A. Existing Primitives, Related Work, and Missing Object

5G systems provide analytics, exposure, policy-control, and application-facing QoS APIs for application–network coordination [6]–[9]. CAMARA Quality on Demand (QoD) provides session-oriented QoS control, and the Service Enabler Architecture Layer for Verticals (SEAL) defines common enabler services for vertical applications over 3rd Generation Partnership Project (3GPP) networks [11]. These primitives expose metrics, events, QoS profiles, and policy-control entry points. They are necessary for capability-aware industrial services, but they do not define the runtime object created when a multi-phase workflow demand is admitted against network capability.

In this paper, the capability envelope is an abstraction built above existing analytics, exposure, and policy-control functions, not a new 3GPP information element. It may be derived from NWDAF analytics, NEF exposure, CAMARA QoD sessions, operator policy, or local admission estimates. We assume such an envelope is available and focus on the software layer needed after a workflow trajectory is admitted. Without an explicit owner, key lifecycle questions remain open: whether the envelope is fresh, whether a retransmission maps to an existing commitment, whether workflow execution may start, and whether a capability-change event should trigger downgrade, upgrade, recovery, release, or abort.

Softwarized service orchestration systems address a different layer of the problem. Network Function Virtualization Management and Orchestration (NFV MANO) frameworks manage network-service, Virtual Network Function (VNF), Cloud-Native Network Function (CNF), slice, and service-level agreement (SLA) lifecycles. Software-Defined Networking (SDN), Multi-access Edge Computing (MEC), cloud-native orchestration, intent-based networking, and zero-touch management steer traffic, place service components, and automate network-service configuration [12]. These systems target network-service or slice desired state; the Broker targets the admitted workflow–network agreement that must remain aligned with capability information, network-side reservation state, and policy-control actions.

The Broker also uses known distributed-systems patterns, including idempotent message handling, long-running sagas, durable state, and state repair [13]–[15]. These patterns support retried requests, long-running workflows, and desired-state repair. A saga orchestrator is a useful analogy for long-running compensation, but it does not validate network-capability freshness, own Network Agent reservation state, or synchronize Broker–Network Agent state after policy-control message loss.

B. Capability Commitment

We consider an industrial workflow composed of ordered phases. Each phase may support multiple QoS modes, such as full, reduced, and fallback modes. A demand trajectory specifies how upcoming phases are mapped to selected QoS modes and fallback options. This trajectory may be generated by an application-side planner, a service orchestrator, or an optimization module.

The network side owns capability estimation, policy constraints, and QoS enforcement. It may provide a capability envelope that describes which QoS profiles are expected to be supportable within a planning window. The envelope is not a permanent guarantee or a standardized reservation. It is valid only within its disclosed range and may change when network conditions or admitted demands change.

Once a demand trajectory is admitted against a capability envelope, the Broker represents the result as a capability commitment. A commitment contains a stable commitment identifier, workflow identifier, referenced envelope, idempotency key, validity time, selected trajectory, lifecycle state, and trace identifier. It is narrower than a full workflow model because it records only the communication-related agreement that the Broker must validate and enforce during execution.

The commitment object separates ownership across the three parties. The Industrial Agent (IA) owns workflow semantics, including phase order, mode alternatives, quality tradeoffs, and fallback tolerance. The Network Agent (NA) owns capability estimation and QoS enforcement. The Broker owns the admitted agreement between them.

C. Control-Plane Failures

Without a lifecycle-managed commitment object, capability-aware workflow orchestration exposes four control-plane failures: stale-envelope admission based on an expired capability view; duplicate commitments caused by retransmitted submissions; illegal transitions such as starting before admission or acknowledging recovery outside the recovery state; and Broker–Network Agent inconsistency after an admission, switch, or release message is lost.

These failures are not data-plane failures. They are software control-plane consistency failures caused by the absence of an explicit owner for the workflow–network agreement. The Broker addresses this gap through four design objectives: commitment ownership, validity-aware policy actions, lifecycle correctness under asynchronous events, and controlled recovery with observability.

We use commitment consistency to mean that an admitted trajectory is fresh, unique under retransmission, lifecycle-legal,

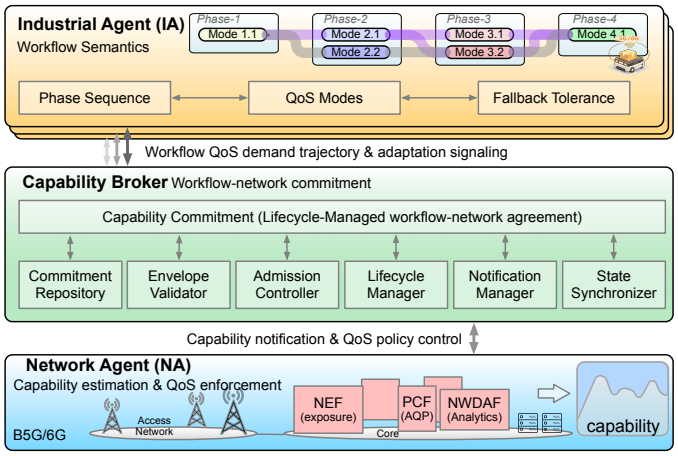

Fig. 1. Broker architecture and control paths.

consistently ordered under concurrent events, and synchronized with the network-side reservation state.

III. Broker Design and Lifecycle Semantics

The Broker is a software control-plane component placed between the IA and the NA, as shown in Fig. 1. It does not replace workflow planning, network analytics, or QoS policy control. It owns the capability commitment defined in Section II and keeps the admitted workflow–network QoS agreement consistent across admission, activation, recovery, release, and abort.

A. Architecture

The architecture follows the ownership split defined in Section II: the IA owns workflow semantics, the NA owns capability estimation and QoS enforcement, and the Broker owns the admitted agreement between them.

The Broker is intended as an application-function or service-orchestration component behind NEF/CAMARA exposure, not as a replacement for standardized 5G core functions. The NA is an adapter for capability estimation and policy-control calls, keeping the Broker outside the PCF/NWDAF/NEF implementation boundary.

In normal operation, the NA publishes or updates a capability envelope. The IA submits a demand trajectory that maps upcoming workflow phases to selected QoS modes and fallback options. The Broker validates the request, invokes network-side admission through the NA, and creates a capability commitment if admission succeeds. Later workflow progress, capability changes, recovery decisions, and release operations are processed through the same commitment object.

This placement keeps the Broker focused. It does not need to know proprietary workflow logic or compute radio-resource allocation. It manages only the control-plane state required to keep the admitted workflow–network QoS agreement consistent across IA events, NA events, and policy actions.

TABLE I
Broker software responsibilities.

Component	Software responsibility	Prevented failure
Commitment Repository	Durable commitment state	Untracked agreement
Envelope Validator	Freshness before policy action	Stale admission or recovery
Admission Controller	Duplicate-safe admission	Duplicate commitment
Lifecycle Manager	Table-driven state transition	Illegal workflow progress
Notification Manager	Event ordering per commitment	Lost-update race
State Synchronizer	State checking and repair	Broker–NA inconsistency

B. Components and Control Semantics

The Broker consists of six logical components. The first five enforce local per-event checks on the synchronous IA–Broker–NA path. The State Synchronizer runs outside this path as a background state-checking and repair module.

The Commitment Repository stores the durable state of each pending or admitted commitment. The Envelope Validator checks whether the referenced capability view is still valid before the Broker invokes any NA-side policy action. The Admission Controller maps retransmitted submissions with the same idempotency key to the existing commitment. The Lifecycle Manager applies a transition table to prevent workflow progress outside the admitted state sequence. The Notification Manager serializes events per commitment so that concurrent workflow-side and network-side events cannot overwrite each other's state updates.

The State Synchronizer handles failures that local checks cannot observe. It compares the Commitment Repository with the NA reservation set and repairs disagreements. In production, NA-only reservations should be checked, retried, or reported before any active QoS flow is removed. Deterministic force-release is used only in the prototype to measure convergence. The local components preserve Broker-side consistency, while the State Synchronizer bounds Broker–NA inconsistency caused by lost NA messages. Periodic repair is sufficient because the Broker owns commitment state and the NA exposes applied reservation state.

The Broker exposes two groups of interfaces. The IA-facing interface receives demand trajectories, workflow start events, phase progress events, adaptation choices, completion events, and abort requests. The NA-facing interface receives capability envelopes, admission results, capability-change notifications, policy activation results, profile-switch results, and release acknowledgements. A demand trajectory first creates a pending commitment candidate. Workflow progress is accepted only if it matches the current commitment state. A capability-change event is routed to recovery instead of directly overwriting the admitted trajectory.

C. Commitment Lifecycle

We model the commitment lifecycle as a finite-state machine (S, E, T). The state set S is shown in Fig. 2, E contains

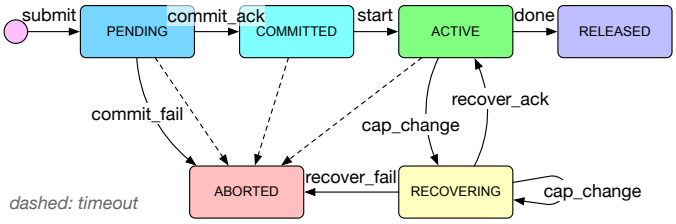

Fig. 2. Commitment lifecycle.

the IA and NA events accepted by the Broker interfaces, and T is the figure's transition relation. Admission is enabled only when the referenced envelope is fresh, the idempotency key is valid, the transition is legal, and the NA admits the requested QoS support.

Fig. 2 shows the lifecycle of a capability commitment. The Broker first creates a commitment in the `Pending` state when the IA submits a demand trajectory. It then validates the referenced capability envelope, checks the idempotency key, and requests network-side admission through the NA. If admission succeeds, `commit_ack` moves the commitment to `Committed`; otherwise, `commit_fail` moves it to `Aborted`.

The `Committed` state means that the trajectory has been admitted and the corresponding network-side QoS support has been prepared. A `start` event moves the commitment to `Active`. In this state, the workflow executes under the admitted trajectory, and phase progress is checked against the commitment. A `done` event moves the commitment to `Released`, where associated QoS policy state is released.

A runtime capability change may affect an admitted segment. Then, `cap_change` moves the commitment from `Active` to `Recovering`. The Broker requests an adaptation decision from the IA, validates it against the latest capability view, and triggers the corresponding QoS policy update through the NA. If recovery succeeds, `recover_ack` returns the commitment to `Active`; otherwise, `recover_fail` moves it to `Aborted`. A `timeout` event may also abort any non-terminal commitment. Both `Released` and `Aborted` are terminal states; later events are ignored safely and logged.

D. Protocol Invariants

The first five components implement a uniform acceptance check. For an event e associated with commitment c in current state s_c, the Broker accepts e only if

$$\begin{aligned}\textsc{Accept}(e,c) = (&\textsc{IsCapChg}(e) \vee \textsc{Fresh}(e.\mathit{env}, e.\mathit{seg})) \\ &\wedge (\neg \textsc{IsAdm}(e) \vee \textsc{Dedup}(e.\mathit{key})) \\ &\wedge \textsc{Legal}(s_c, e),\end{aligned} \tag{1}$$

where IsCapChg identifies capability-change notifications, IsAdm identifies admission requests, Fresh is the Envelope Validator's freshness check on the referenced segment, Dedup is the Admission Controller's idempotency-key lookup, and Legal is the Lifecycle Manager's transition-table check. If $\textsc{Accept}(e,c)$ is false, the Broker rejects the event before issuing any NA-side policy action. The Notification Manager guarantees that this evaluation is atomic per commitment, so two concurrent events on the same c cannot both observe the same s_c and write incompatible post-states.

Together, the finite-state machine and acceptance predicate specify the Broker's consistency guarantee: every accepted event is fresh when it triggers a policy-control action, duplicate-safe when it creates a commitment, legal under the lifecycle relation, and atomic with respect to other events on the same commitment.

Three invariants follow. **I1: No policy action from stale capability.** No admission, recovery, or profile-switch action reaches the NA unless Fresh holds for the affected segment at the moment of action. The `cap_change` event is exempt from this check because it signals that the envelope has changed; freshness is checked again at `recover_ack`, after the IA submits an adaptation that fits the new envelope.

I2: One commitment per idempotency key. For a given idempotency key, Dedup returns the existing commitment instead of creating a new one [13]. A retransmission whose trajectory fingerprint differs from the stored one is rejected as a key collision.

I3: Legal state progression. A commitment state can change only through an event that is legal under the lifecycle transition table. In particular, a capability-change event first routes the commitment to `Recovering` instead of directly changing the admitted trajectory.

The acceptance predicate is local. It cannot detect that an earlier NA-side action was lost. The State Synchronizer complements it with a state repair loop [15], which is evaluated in Section IV.

IV. Prototype and Evaluation

A. Prototype and Workloads

The evaluation focuses on control-plane consistency rather than radio-resource efficiency. The Broker is designed to keep an admitted workflow–network QoS agreement consistent under stale capability, retransmission, asynchronous events, and message loss toward the Network Agent.

We implement the Broker prototype in Python with an IA driver, an NA surface, a Commitment Repository, an Envelope Validator, an Admission Controller, a Lifecycle Manager, a Notification Manager, and a State Synchronizer. The IA submits demand trajectories and workflow progress events, while the NA publishes capability envelopes, returns admission decisions, and receives QoS policy calls for admission, activation, profile switching, and release.

We validate the prototype in a hybrid setup. The NA surface is aligned to the Fraunhofer Institute for Open Communication Systems (FOKUS) 5G test platform [16], while the IA is simulated through our digital twin box [17]. We use real `pcf_applied` latency samples collected on that platform to calibrate the order of magnitude of the NA-side policy delay. Ten samples from a four-phase profile schedule show two regimes: warm-cache profile changes complete in 62–214 ms, while cold-cache profile installation requires 2.18–

2.31 s. These samples are not used as a statistical benchmark of the FOKUS platform. They show that the NA-side policy delay is much larger than the Broker-side processing prefix.

The consistency experiments run in the Broker harness with deterministic IA and NA callbacks; the 5G platform measurements calibrate the policy-control delay scale. The reported loads are fixed-size control-plane workloads: 10,000 invocations per local benchmark operation, a 10,000-operation mixed ablation trace, and 1,000 commitment lifecycles for the loss study. Deterministic callbacks allow retransmission, expired envelopes, illegal lifecycle events, concurrent workflow/network events, and controlled message loss to be reproduced. A trace replay confirms that the full Broker follows the expected path `Pending` → `Committed` → `Active` → `Recovering` → `Active` → `Released` under a three-phase workflow with one capability-change event. The same replay verifies that a capability-change event is routed through `Recovering` instead of directly changing the admitted trajectory.

B. Local Overhead

We first measure local Broker overhead for admission processing, envelope validation, idempotency lookup, lifecycle transition checking, and event dispatch. Each operation is invoked 10,000 times under 1, 10, and 100 active commitments. Latency is measured using a monotonic high-resolution timer.

Across all three load levels, admission processing measures 5.4–5.8 μs at the 50th percentile (P50) and 7.5–7.6 μs at the 95th percentile (P95). Event dispatch measures 0.88–0.96 μs P50. The three pure checks, namely envelope freshness, idempotency lookup, and lifecycle transition, each complete in 0.08–0.17 μs P50. None of the five operations grows with the number of active commitments because the internal lookups are hash-indexed.

These local costs are several orders of magnitude smaller than the calibrated NA-side policy delay. The Broker's per-admission prefix of approximately 5.5 μs is about $4 \times 10^{-3}\%$ of the warm-cache median and $2.5 \times 10^{-4}\%$ of the cold-cache median. Under the measured policy-delay regime, the Broker does not dominate end-to-end commitment latency.

C. Fault-Injection Test

The main evaluation tests whether each Broker function prevents its corresponding failure class. This is a fault-injection test, not a claim that the injected failure rates match real deployment rates. We construct a 10,000-operation mixed workload with deterministic randomization. The workload contains 5,000 normal lifecycle events, 1,500 retransmitted submissions sharing an idempotency key, 1,500 submissions referencing an expired envelope, 1,000 illegal-transition probes from `Committed`, and 1,000 concurrent race operations on active commitments. Each illegal-transition probe attempts three illegal events, and each race operation issues a concurrent `cap_change`/`done` pair.

We compare the full Broker with four ablated variants:

TABLE II
FAULT-INJECTION ABLATION RESULTS.

Variant	Duplicate	Stale	Illegal	Divergence
No Idempotency	1500	0	0	0
No Envelope Validator	0	1500	0	0
No Lifecycle Manager	0	0	3000	1000
No Notification Manager	0	0	0	945
Full Broker	0	0	0	0

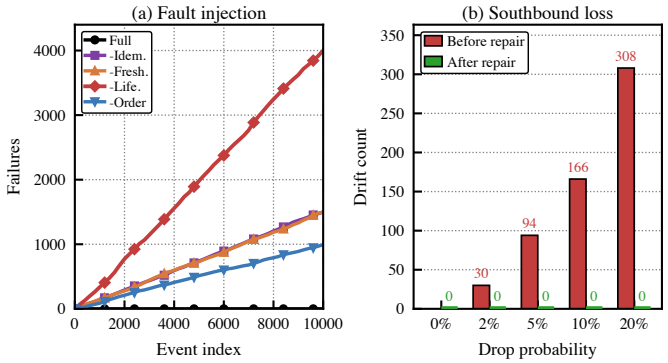

Fig. 3. Consistency under injected faults and message loss.

- **No Idempotency**: retransmitted demand submissions are not deduplicated.
- **No Envelope Validator**: envelope validity is not checked before admission or recovery.
- **No Lifecycle Manager**: events are applied without checking the legal transition table.
- **No Notification Manager**: concurrent workflow-side and network-side events are not serialized per commitment.

We measure four failure classes. A *duplicate commitment* occurs when the same idempotency key creates more than one commitment. A *stale admission* occurs when a trajectory is admitted against an expired envelope. An *illegal transition* occurs when an out-of-table lifecycle transition is accepted. A *state divergence* occurs when concurrent events produce inconsistent workflow-side and network-side interpretations of the commitment state.

Table II and Fig. 3(a) show that each removed function exposes its associated failure class. Without idempotency, retransmissions create duplicate commitments. Without envelope validation, expired-envelope submissions become stale admissions. Without lifecycle validation, each illegal-transition probe accepts three illegal events, giving 3,000 illegal transitions; concurrent race operations add 1,000 state divergences. Without event ordering, the unsafe variant exposes 945 lost-update divergences, where racing events read the same pre-state and write incompatible post-states.

The full Broker rejects all injected failures and preserves commitment consistency throughout the workload. The divergence counts reflect the intentionally opened race windows in the ablated variants, not real deployment rates.

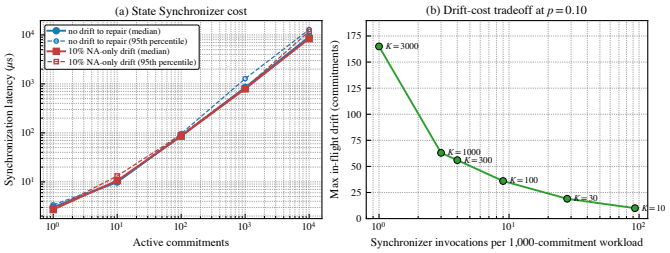

Fig. 4. State Synchronizer cost and temporary inconsistency.

D. Broker–NA Inconsistency and State Synchronization

The previous experiments assume that the NA receives all Broker-issued policy calls. In practice, messages toward the NA may be delayed or lost before the Broker observes a successful network-side state change. We measure inconsistency at the end of the workload, after every commitment has been driven through admission, start, and completion. A commitment is inconsistent if the Broker and NA disagree about whether its reservation is still held.

We evaluate this effect with an NA simulator that maintains its own reservation table and drops both request and acknowledgement messages with probability p applied symmetrically to each leg. We sweep $p \in \{0, 0.02, 0.05, 0.10, 0.20\}$. At each loss level, we replay a 1,000-commitment lifecycle workload and measure inconsistency twice: once after the workload, and once after a single State Synchronizer pass.

Fig. 3(b) shows that without state synchronization, inconsistency grows with the loss rate: 0, 30, 94, 166, and 308 inconsistent commitments out of 1,000 at $p = 0\%$, 2%, 5%, 10%, and 20%, respectively. At the same loss levels, the effective admission success rate falls from 100% to 96.8%, 91.8%, 82.4%, and 66.3% because reservation requests or acknowledgements are silently dropped. State synchronization does not recover lost admission opportunities, but it exposes and repairs residual state disagreement. After one State Synchronizer pass, inconsistency drops to zero at every tested loss level; a second immediate pass detects zero residual inconsistency.

E. State Synchronization Cost and Tunability

The previous experiment runs state synchronization only at the end of the workload, whereas a deployed Broker would invoke it periodically. We evaluate both scalability and tunability. For scalability, we time 200 synchronizer invocations for $N \in \{1, 10, 100, 1{,}000, 10{,}000\}$ active commitments, with either no inconsistency or 10% NA-only inconsistency. For tunability, we replay the 1,000-commitment workload at $p = 0.10$ and trigger synchronization every $K \in \{10, 30, 100, 300, 1{,}000, 3{,}000\}$ events.

Fig. 4(a) shows approximately linear scaling: P50 latency rises from $3\,\mu$s at $N = 1$ to about 9 ms at $N = 10{,}000$, with P95 within about $1.5\times$ of P50. Fig. 4(b) shows that $K = 100$ keeps maximum temporary inconsistency at most 36 commitments with 9 invocations over the workload, smaller K reduces inconsistency further at higher background cost.

Overall, the results show that the Broker's local checks add lightweight processing overhead, while its semantic checks and synchronizer prevent the main control-plane consistency failures introduced by stale capability, retransmission, illegal lifecycle events, concurrent notifications, and lossy signaling toward the NA. The prototype isolates Broker-side consistency from radio-resource scheduling and assumes one Broker instance. Production deployment requires integration with NEF/PCF/NWDAF and replicated Broker state.

V. Conclusion

This paper identified workflow–network QoS coordination as a missing commitment-management function in the software control plane of capability-aware industrial B5G/6G services. The proposed Capability Broker represents each admitted trajectory as a lifecycle-managed QoS commitment and enforces freshness, duplicate-safe admission, legal state progression, per-commitment ordering, recovery routing, and Broker–network state synchronization. Prototype results show that removing Broker semantics exposes duplicate commitments, stale admissions, illegal transitions, lost-update divergences, or Broker–NA inconsistency, while the full Broker preserves commitment consistency with microsecond-level local checks and a linearly scalable State Synchronizer. Future work will extend the Broker toward multi-domain coordination, probabilistic capability envelopes, durable Network-Agent signaling, replicated deployment, and joint management of communication and edge-computing resources.

Acknowledgment

This work was supported in part by the German Federal Ministry of Research, Technology and Space (BMFTR) through 6GEM+ under Grant 16KIS2411; and by the European Union's Horizon Europe programme through 6G-Path (Grant No. 101139172) and 6GSandbox (Grant No. 101096328). This research was also partially conducted at ICTFICIAL Oy. The paper reflects only the authors' views, and the European Commission bears no responsibility for any utilization of the information contained herein.

References

[1] 5G-PPP Technology Board, "Innovation Trends in I4.0 enabled by 5G and Beyond Networks," 5G-PPP Technology Board, White Paper, Oct. 2023.

[2] 5G Alliance for Connected Industries and Automation, "5G for Automation in Industry," 5G-ACIA, White Paper, 2019.

[3] P. Varga *et al.*, "5G Support for Industrial IoT Applications," *IEEE Communications Magazine*, 2020.

[4] C. Xu, W. Jiang, H. Ma, Y. Liu, H. Men, X. Xu, and Z. Li, "A review: Research and application of pipeline robots in the oil and gas industry," *Journal of Pipeline Science and Engineering*, p. 100356, 2025.

[5] L. Meitz, J. Senge, T. Wagenhals, T. Schöler, J. Hähner, J. Edinger, and C. Krupitzer, "A literature review framework and open research challenges for predictive maintenance in industry 4.0," *Computers & Industrial Engineering*, vol. 206, p. 111193, 2025.

[6] 3GPP, *5G System; Network Exposure Function Northbound APIs; Stage 3*, 3rd Generation Partnership Project (3GPP) Technical Specification TS 29.522, Sep. 2024, nEF northbound exposure APIs.

[7] 3GPP, *Architecture enhancements for 5G System (5GS) to support network data analytics services*, 3rd Generation Partnership Project (3GPP) Technical Specification TS 23.288, Jun. 2024, predictive QoS via NWDAF analytics.
[8] 3GPP, *Policy and Charging Control Framework for the 5G System (5GS); Stage 2*, 3rd Generation Partnership Project (3GPP) Technical Specification TS 23.503, Sep. 2024, alternative QoS Profile (AQP) mechanism.
[9] CAMARA Project, "Quality on demand (QoD) API," Linux Foundation, GSMA Open Gateway, 2024.
[10] Q. Guo, Y. Chen, T. Taleb, B. Riemer, H. Zope, and H. Yu, "Beyond per-request qos: Coordinating industrial workflows with b5g/6g network capabilities," *arXiv preprint arXiv:2605.00570*, 2026.
[11] 3GPP, *Service Enabler Architecture Layer for Verticals (SEAL); Functional architecture and information flows*, 3rd Generation Partnership Project (3GPP) Technical Specification TS 23.434, Apr. 2024.
[12] ETSI, *Zero-touch network and Service Management (ZSM); Closed-Loop Automation; Part 1: Enablers*, European Telecommunications Standards Institute Group Specification GS ZSM 009-1, Jun. 2021.
[13] P. Helland, "Idempotence is not a medical condition," *Communications of the ACM*, vol. 55, no. 5, pp. 56–65, May 2012.
[14] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987, pp. 249–259.
[15] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95, 1995, pp. 172–183.
[16] B. Riemer, P. Merino, and M. Dieudonne, "Berlin platform; 6G-SANDBOX SNS HE Project," https://6g-sandbox.eu/pilot-6g-sites/berlin/, 2026, [Accessed 18-03-2026].
[17] Q. Guo, Y. Chen, T. Taleb, and C. Yang, "Enabling modular and intelligent iort systems: Integrating the model context protocol for semantic decoupling at the edge," *IEEE Internet of Things Magazine*, vol. 9, no. 1, pp. 39–46, 2026.