

que formam a interface do receptor. O ecossistema adotado pelo *testbed* favorece naturalmente essa extensão: o MQTT desacopla os serviços de *back-end*, o Redis persiste o estado de governança de forma consultável, e a abordagem de *microfrontends* integra componentes visuais independentes.

Embora a arquitetura proposta estabeleça uma base funcional para experimentação, ela apresenta lacunas que restringem sua adoção em cenários com múltiplos grupos de pesquisa e com requisitos de conformidade à norma. Os autores do *testbed* indicam, como trabalho futuro, a necessidade da “*adoção de um broker com suporte à validação de esquemas, visando padronizar a comunicação e aumentar a robustez da arquitetura distribuída*” [9], bem como uma análise comparativa de *brokers* segundo critérios de consumo de recursos, suporte à validação de *payloads*, facilidade de configuração e disponibilidade de *plugins* [9, 10]. Adicionalmente, a análise da implementação de referência disponível [10] permite identificar três outras lacunas relevantes para a conformidade do *testbed* com a ABNT NBR 25608. A primeira é a inexistência de controle de acesso diferenciado por classe de cliente (aqui, *cliente* designa a aplicação que se conecta ao *broker* MQTT, e não o telespectador, usuário humano dono de um perfil no receptor), embora a norma estabeleça distinção entre aplicações internas ao receptor, aplicações externas na rede doméstica e *broadcasters* [5]. A segunda é a falta de um mecanismo de consentimento verificável em tempo de execução, necessário para que o Gerenciador de Privacidade previsto na norma, alinhado à ISO/IEC 27560:2023 [1], se traduza em restrições efetivas ao acesso a dados de perfil do telespectador. A terceira é a dependência de *scripts* de inicialização específicos de sistema operacional, o que dificulta a reprodução do ambiente por pesquisadores em configurações heterogêneas e contraria o objetivo declarado de “*disseminação e adoção em larga escala*” do *testbed* [9].

Este trabalho preenche algumas lacunas citadas anteriormente entregando contribuições em cada uma das três camadas. Isso demonstra a extensibilidade do *testbed*.

Na camada de *back-end*, um *plugin* nativo em C para o *broker* Mosquitto executa, em sequência, controle de acesso por lista de clientes, verificação de consentimento por serviço, conforme a ABNT NBR 25608 e a ISO/IEC 27560, e validação de esquema JSON dos *payloads*. Na camada de WebServices, dois gateways de API com papéis distintos centralizam a autenticação via JWT e a verificação de consentimento para clientes externos e internos. Na camada visual, um novo *microfrontend* de criação de perfis de telespectador demonstra que a interface pode ser estendida sem modificação dos componentes existentes. Todo o ambiente é containerizado via Docker Compose, substituindo *scripts* de inicialização específicos de sistema operacional por uma topologia reproduzível com um único comando. O esforço de configuração do ambiente é avaliado por inspeção, em comparação com a arquitetura original.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta a arquitetura do *testbed* original que este trabalho estende. A Seção 3 discute os trabalhos relacionados. A Seção 4 descreve as extensões propostas. A Seção 5 apresenta a avaliação da reprodutibilidade do ambiente. A Seção 6 discute os resultados e as limitações. Finalmente, a Seção 7 conclui o trabalho.

2 TESTBED PARA TV 3.0

Esta seção resume a arquitetura do *testbed* aberto para TV 3.0 proposta em [9, 10], estabelecendo uma base para a apresentação das extensões propostas neste trabalho.

2.1 Arquitetura Distribuída da AoP

No *front-end*, a arquitetura adota a abordagem de *microfrontends*: a interface do receptor é composta por componentes visuais independentes, como o catálogo de aplicações, o seletor de perfis e a aplicação *bootstrap*, carregados dinamicamente conforme o estado publicado no *broker* [10].

A arquitetura proposta em [9] é baseada em microsserviços e segue o padrão *publish/subscribe* com um *broker* MQTT centralizado. Cada componente da AoP é implementado de forma independente e compartilha o estado da plataforma por meio de tópicos MQTT retidos no *broker*. A Figura 2 ilustra o ecossistema de componentes distribuídos. Nele estão presentes o servidor AoP, o componente de TV 3.0 WebServices, o *broadcaster* e os componentes de aplicações Ginga.

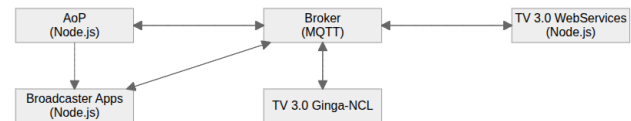


Figure 2: Ecossistema distribuído de componentes da AoP. Adaptado de [9].

A estrutura de tópicos MQTT reflete o estado da plataforma, conforme definido em [9]. Os tópicos abrangem desde a identificação do serviço e aplicação em execução, como em `aop/currentService` e `aop/<id>/path`, até o estado de máquinas de eventos de nós Ginga-NCL, como em `.../presentationEvent/state` e `.../attributionEvent/value`. Há também tópicos para dispositivos remotos registrados, como `aop/devices/<class>`. Todos os tópicos com dados de estado são retidos no *broker*, sendo reenviados automaticamente a novos assinantes.

2.2 Casos de Uso Validados

A utilidade da arquitetura foi demonstrada por dois casos de uso [10]. No primeiro, o motor de apresentação Guaraná [19] registra-se como dispositivo remoto da classe `nc1-360-vr` via *Remote Devices API* do TV 3.0 WebServices. A partir desse registro, o componente Ginga-NCL associa cada nó `nc1360` da aplicação a um dispositivo conectado e o TV 3.0 WebServices traduz as notificações de ação em metadados JSON enviados ao Guaraná via WebSocket, convertendo de volta em publicações os eventos recebidos do HMD. A inicialização completa de uma cena 360 foi medida em seus estágios de comunicação, sendo o tempo dominado pela preparação da cena no Guaraná (1.801 ms), enquanto a posterior alteração de estado de um objeto de mídia na cena já carregada permanece abaixo de poucos milissegundos [10]. No segundo caso, a *Sensory Effect Presentation Engine* (SEPE) registra-se como dispositivo remoto da classe `sensory-effect`. A SEPE traduz comandos genéricos de efeitos sensoriais em protocolos proprietários de dispositivos IoT, especificamente uma lâmpada Xiaomi Yeelight controlada via TCP local

e um difusor Moodo controlado via API em nuvem. As latências medidas foram de 8,97 ms para efeitos de iluminação e 559,27 ms para efeitos olfativos. Ambas estão dentro dos limites aceitáveis para as respectivas modalidades [13].

2.3 Limitações Identificadas

Os trabalhos anteriores [9, 10] identificam como trabalhos futuros a adoção de um *broker* com suporte à validação de esquemas de *payloads*, visando incrementar a robustez da plataforma e impedindo que o estado armazenado no *broker* se torne inconsistente; a análise de *brokers* alternativos com base em critérios de consumo de recursos, facilidade de configuração e disponibilidade de *plugins*, visando suportar, e.g., a validação de *payloads*; e a inclusão de mais funcionalidades da AoP, como a criação completa de perfis de telespectadores. A ausência de controle de acesso diferenciado por cliente e de verificação de consentimento em tempo de execução também constituem limitações identificadas na análise da implementação da API de usuários conforme a ABNT NBR 25608 [5]. Este trabalho endereça parte dessas limitações com o objetivo central de demonstrar o reúso e a extensibilidade do *testbed*, reduzindo o esforço necessário para que novos desenvolvedores estendam a plataforma. As contribuições apresentadas evidenciam essa extensibilidade em cada camada: o *plugin* de governança (Seção 4.2) trata, no *back-end*, a validação de esquemas de *payloads*, o controle de acesso por cliente e a verificação de consentimento em tempo de execução; e o *microfrontend* de criação de perfis (Seção 4.4) estende a camada visual, ambos integrados sem modificação dos componentes existentes. A análise comparativa de *brokers* alternativos permanece em aberto, uma vez que este trabalho opta por estender o Mosquitto em vez de substituí-lo, conforme discutido na Seção 3.

3 TRABALHOS RELACIONADOS

A validação de *payloads* em *brokers* MQTT tem sido tratada na literatura de IoT predominantemente pela substituição do *broker* por soluções com registro de esquemas nativo. O EMQX [11] e o HiveMQ Enterprise oferecem suporte a JSON Schema, Avro e Protobuf em suas versões comerciais, com regras de roteamento e rejeição configuráveis. Contudo, introduzir um *broker* distinto em um *testbed* existente implica aprendizado de uma nova plataforma e possível incompatibilidade com a comunidade acadêmica brasileira de TV digital, que tem histórico de uso do Mosquitto. A abordagem adotada neste trabalho mantém o Mosquitto e o estende por meio de um *plugin* C, preservando a superfície familiar enquanto adiciona governança de dados. Essa decisão alinha-se à recomendação dos autores do *testbed* original de realizar análise comparativa de *brokers* antes de eventual migração [9].

No domínio de TV digital, trabalhos recentes abordam a validação de conformidade de *middleware* e de APIs REST em sistemas DTV brasileiros. Maia et al. [16] propõem um analisador em tempo real para o módulo de WebServices do Ginga CC, reduzindo em 66,84% o tempo de execução de testes frente ao processo manual. Da Silva et al. [7] apresentam uma metodologia automatizada para testes de conformidade e robustez de pilhas de processamento multimídia em receptores DTV. Sua abordagem emprega tabelas de causa-efeito e *fuzzing* baseado em gramática. Esses trabalhos focam

em verificação *a posteriori* de implementações existentes. A presente proposta é complementar, atuando preventivamente durante a execução do *testbed*.

Ahmed and Bures [2], Ahmed et al. [3] discutem ainda os desafios específicos do teste de aplicações para Smart TV. Esses desafios incluem a fragmentação de plataformas e a navegação direcional baseada em foco. Os autores propõem soluções modulares baseadas em separação de modelos de navegação e funcionalidade. Embora distintos do escopo deste trabalho, esses estudos reforçam a importância de arquiteturas modulares para o ecossistema de TV digital, característica preservada pela presente proposta.

O controle de acesso e consentimento em sistemas de TV interativa é abordado indiretamente na literatura de TV 3.0. A própria ABNT NBR 25608 [5] especifica um Gerenciador de Privacidade alinhado à ISO/IEC 27560 [1], que define estrutura de registros de consentimento. A tradução dessas especificações em mecanismos verificáveis em tempo de execução não foi endereçada em implementações abertas até o presente momento, o que representa uma das principais contribuições deste trabalho no domínio de conformidade normativa.

A reprodutibilidade de ambientes de experimentação em sistemas distribuídos é tema consolidado na literatura de DevOps e computação em nuvem, com Docker Compose como padrão de fato para orquestração de ambientes de desenvolvimento reproduzíveis [8]. No contexto do *testbed* da TV 3.0 AoP, a containerização completa dos componentes traz duas vantagens diretas: elimina a dependência de *scripts* de inicialização específicos de sistema operacional, e padroniza o ambiente de execução entre diferentes grupos de pesquisa que utilizem a plataforma.

A integração de componentes de interface independentes em uma única aplicação é tratada na literatura pela abordagem de *microfrontends* [12], que aplica ao *front-end* os princípios de desacoplamento e desenvolvimento independente característicos dos microsserviços. A arquitetura original do *testbed* já adota essa abordagem na camada visual, e este trabalho a estende com um novo componente de criação de perfis, conforme será descrito na Seção 4.4.

4 ARQUITETURA ESTENDIDA

A arquitetura estendida proposta neste trabalho adiciona elementos a cada camada de extensão do *testbed* original [9, 10], sustentados por um modelo de dados comum. A Seção 4.1 descreve esse modelo, que combina o *broker* MQTT e um *store* Redis persistente. A Seção 4.2 apresenta o *plugin* de governança nativo ao *broker* MQTT, que atua sobre o barramento de comunicação entre os serviços de *back-end*. A Seção 4.3 descreve os dois *gateways* de API, na camada de WebServices. A Seção 4.4 apresenta a extensão da camada visual por meio de um novo *microfrontend*. A Seção 4.5 descreve a infraestrutura containerizada que integra todos esses componentes.

4.1 Modelo de Dados Compartilhado

A arquitetura original [9, 10] mantém todo o estado da AoP em tópicos MQTT retidos no *broker*. Esse modelo é adequado para o estado observável da plataforma, que precisa ser propagado em tempo real aos componentes assinantes. São exemplos desse estado

o perfil ativo, o serviço em execução e a lista de dispositivos remotos. Contudo, para suportar as decisões de governança propostas neste trabalho, o modelo precisa ser estendido. O *plugin* do *broker* precisa consultar, no caminho crítico de cada publicação, estruturas de dados que não são naturalmente expressas por mensagens em tópicos, como conjuntos de padrões de tópico autorizados por cliente, conjuntos de serviços com consentimento concedido por usuário e esquemas JSON associados a padrões de tópico.

Essas consultas ocorrem no caminho crítico de cada publicação, o que impõe três requisitos ao mecanismo de armazenamento: (i) acesso síncrono e de baixa latência, já que o *plugin* não pode aguardar a entrega assíncrona de mensagens retidas; (ii) operações de pertinência a conjunto com custo $O(1)$, exigidas pela verificação de ACL e de consentimento a cada mensagem; e (iii) persistência em disco, para que o estado de governança sobreviva a reinícios. O modelo de tópicos retidos do MQTT não atende (i) e (ii), pois expõe estado por notificação e não por consulta direta indexada. Optou-se pela utilização do Redis [18] por satisfazer os três requisitos simultaneamente: estruturas consultáveis por chave com pertinência $O(1)$ (e.g., `SISMEMBER`), persistência configurável em modo AOF (*Append-Only File*) e provisionamento como um único contêiner.

A solução adotada é a coexistência de dois *stores* persistentes. O Mosquitto mantém persistência das mensagens retidas em disco, conforme configuração padrão do *broker*, garantindo que o estado observável da AoP sobreviva a reinícios. O Redis [18] é introduzido como *store* chave-valor persistente para o estado de governança e conformidade, com persistência configurada no modo AOF com sincronização a cada segundo, oferecendo durabilidade comparável à de um sistema de banco de dados relacional. A ideia central é que os tópicos retidos do MQTT que representam estado consultado pelo *plugin* sejam materializados como estruturas correspondentes no Redis, permitindo que esse estado continue disponível como canal de notificação para componentes assinantes e, simultaneamente, como estrutura consultável de forma síncrona pelo *plugin*. Dessa forma, MQTT e Redis cumprem papéis complementares e não duplicados: o MQTT é canal de notificação por tópico, com semântica *publish/subscribe*, enquanto o Redis é *store* estruturado consultável por chave, com semântica de acesso direto.

A Tabela 1 resume o mapeamento entre tópicos MQTT e estruturas Redis utilizado neste trabalho. O *hash* `user:{id}` mantém os campos do perfil de telespectador previstos na Tabela 7 da ABNT NBR 25608, incluindo *nickname*, *avatar*, controles parentais e atributos de acessibilidade. O *hash* `user:{id}:broadcaster-attrs:{serviceContextId}` mantém atributos do usuário específicos ao contexto de um serviço de radiodifusão, conforme a Seção C.6.16 da norma. A consistência entre os dois *stores* é mantida pelo componente de WebServices, que atua como ponto único de escrita: ao receber uma operação que altera o estado da AoP, o componente atualiza a estrutura correspondente no Redis e em seguida publica a mensagem retida no tópico MQTT. Operações de leitura no caminho crítico do *plugin* consultam exclusivamente o Redis, evitando a complexidade de embarcar um cliente MQTT em código C nativo e a latência associada a aguardar a entrega de mensagens retidas.

Tópico MQTT	Estrutura Redis
<code>aop/currentUser</code>	<code>session:current-user</code> (string)
<code>aop/currentService</code>	<code>session:current-service-id</code> (string)
<code>aop/users</code>	<code>users:index</code> (set) + <code>user:{id}</code> (hash)
(somente Redis)	<code>user:{id}:consent</code> (set)
(somente Redis)	<code>user:{id}:broadcaster-attrs:{serviceContextId}</code> (hash)
(somente Redis)	<code>acl:{clientId}</code> (set)
(somente Redis)	<code>schema:{pattern}</code> (string)

Table 1: Mapeamento parcial para exemplo entre tópicos MQTT e estruturas Redis no modelo de dados compartilhado.

4.2 Plugin de Governança no Broker MQTT

O *plugin* é implementado como uma extensão nativa em C para o *broker* Mosquitto. Ele intercepta o caminho de publicação por meio do *callback* `on_message`, antes que qualquer *payload* seja retido ou entregue a assinantes. Para cada mensagem recebida, o *plugin* executa três verificações em sequência, consultando as estruturas Redis descritas na Seção 4.1. A primeira é o controle de acesso, que verifica se o padrão do tópico publicado pertence ao conjunto autorizado para o cliente em `acl:{clientId}`. A segunda é a verificação de consentimento por serviço, que consulta o conjunto `user:{userId}:consent` para confirmar que o usuário ativo autorizou o serviço associado ao tópico, em alinhamento ao Gerenciador de Privacidade da ABNT NBR 25608 e à estrutura de registros de consentimento da ISO/IEC 27560 [1]. A terceira é a validação de esquema JSON do *payload* contra o esquema cadastrado para o padrão de tópico correspondente em `schema:{pattern}`.

A validação de esquema é implementada por um módulo independente, denominado `schema_validator.c`. O módulo recebe um objeto JSON-C como dado e um objeto JSON-C como esquema, devolvendo uma estrutura `validation_result` com indicador de validade e mensagem de erro. O validador implementa um subconjunto do vocabulário de validação do JSON Schema, conforme a especificação *draft 2020-12* [14], com validação recursiva de estruturas aninhadas, o que permite a especificação de esquemas para estruturas como os vetores de nós e interfaces da Tabela 1 de [9]. A Tabela 2 relaciona os operadores implementados, agrupados por categoria. Operadores de composição lógica e referências entre esquemas não são suportados, por não serem requeridos pelos esquemas dos tópicos da AoP; a implementação completa está disponível no repositório¹.

Os esquemas são armazenados no Redis com chaves no formato `schema:{pattern}`, onde `pattern` é um padrão de tópico MQTT. O *plugin* carrega os padrões cadastrados na inicialização e realiza *matching* local a cada publicação. A política adotada é de *allow-by-default* para tópicos sem esquema cadastrado, o que permite a adoção incremental da validação sem interromper componentes existentes. Quando um *payload* não passa na validação, a publicação é rejeitada e o erro é registrado no *log* do *broker*, contendo o identificador do cliente, o tópico e a descrição do erro retornada pelo validador.

¹<https://github.com/multisens/TV30>

Categoria	Operadores
Genéricos	type, enum, const
Numéricos	minimum, maximum, exclusiveMinimum, exclusiveMaximum, multipleOf
Strings	minLength, maxLength, pattern
Arrays	items, minItems, maxItems, uniqueItems
Objetos	properties, required, additionalProperties, minProperties, maxProperties

Table 2: Operadores do JSON Schema (draft 2020-12) implementados pelo validador.

4.3 Gateways de API

Clientes da rede doméstica, como aplicações *companion* e dispositivos remotos, e os componentes internos do *testbed* acessam a mesma camada de WebServices, mas precisam de tratamento distinto: requisições externas precisam ser autenticadas antes de acessar a plataforma, enquanto chamadas entre componentes confiáveis têm a identidade estabelecida pelo próprio ambiente containerizado e dispensam esse passo. Implementar a autenticação dentro de cada componente espalharia essa lógica pelo *testbed*; por isso ela é centralizada em *gateways* de API, que a aplicam de forma declarativa e externa aos componentes de domínio. Optou-se pelo KrakenD [15] por três características exigidas pelo ecossistema do *testbed*: configuração declarativa por arquivo, que mantém a política versionável junto à infraestrutura; suporte a *plugins* nativos em Go, que permite validar os *tokens* JWT no próprio *gateway* e estender essa verificação a outras políticas baseadas em JWT; e geração de documentação das rotas a partir da própria configuração, mantendo especificação e implementação sincronizadas.

O *gateway* externo é o único ponto de entrada para clientes na rede doméstica, como aplicações *companion* e dispositivos remotos. Ele é equipado com um *plugin* escrito em Go que valida o formato e a validade dos *tokens* JWT das requisições antes de encaminhá-las ao componente interno correspondente, e que pode ser estendido a outras verificações baseadas em JWT. O *gateway* interno atende chamadas entre componentes confiáveis do *testbed*, sem essa validação, uma vez que a identidade do cliente é estabelecida pelo próprio ambiente containerizado.

O fluxo de autenticação segue o padrão de dois passos da ABNT NBR 25608. O cliente solicita uma autorização que exibe um desafio na interface do receptor, na forma de QR code ou PIN. Após resolução, o cliente obtém um par de *tokens* (acesso e renovação) via JSON Web Token (JWT). Os *tokens* de acesso são validados pelo *gateway* em todas as rotas protegidas antes de encaminhar a requisição ao componente interno correspondente.

A documentação das rotas é gerada dinamicamente pelo *middleware* a partir do arquivo de configuração do *gateway*, e disponibilizada via Swagger UI. Essa abordagem mantém documentação e configuração sincronizadas, eliminando a necessidade de manutenção manual de especificações OpenAPI.

4.4 Extensão da Camada Visual via Microfrontend

A camada visual da AoP é construída segundo a abordagem de *microfrontends*, na qual componentes de interface independentes são integrados em uma única aplicação *web* [12]. No *testbed* original, essa camada contempla o catálogo de aplicações, o seletor de perfis e a aplicação *bootstrap*, cada um carregado dinamicamente conforme o estado publicado no *broker*. Para demonstrar a extensibilidade dessa camada, este trabalho adiciona um novo *microfrontend* de criação de perfis de telespectador.

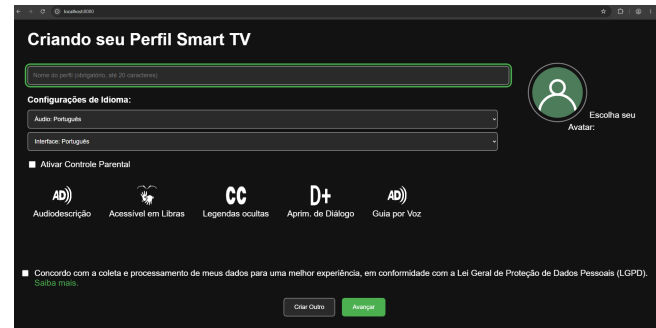


Figure 3: Tela de criação de perfis de telespectador.

O componente adicionado é a tela de criação de perfis, ilustrada na Figura 3. Ela apresenta um formulário aderente aos campos definidos na Tabela 7 da ABNT NBR 25608, incluindo apelido, avatar, controle parental, classificação etária máxima e atributos de acessibilidade. Ao submeter o formulário, o componente invoca o *end-point* de criação de perfil exposto pelo componente de WebServices, que valida os campos requeridos, persiste o novo perfil no Redis e publica a atualização no *broker*, de modo que os demais *microfrontends* sejam notificados e atualizem suas visões. A adição desse componente não exigiu modificações nos *microfrontends* existentes, evidenciando que a camada visual pode ser estendida de forma independente, característica essencial para o desenvolvimento colaborativo da plataforma.

4.5 Infraestrutura Containerizada

A infraestrutura do *testbed* é orquestrada por um arquivo Docker Compose que define todos os serviços que compõem a arquitetura estendida: o *broker* Mosquitto com o *plugin* de governança, o Redis, os dois *gateways* KrakenD, os *middlewares* de autenticação, o servidor AoP, o componente de WebServices e o componente *broadcaster*. Todos os serviços são executados em *containers* conectados por uma rede interna dedicada, e o provisionamento completo do ambiente é realizado com um único comando, eliminando a dependência de *scripts* de inicialização específicos de sistema operacional presente na arquitetura original. O *testbed* destina-se à execução local, em um único *host*, como ambiente de estudo e experimentação, e não a implantações em produção; as decisões de configuração e de segurança descritas neste trabalho assumem esse contexto. Todos os serviços são executados em *containers* conectados por uma rede interna dedicada. A Figura 4 ilustra a arquitetura estendida, destacando sobre o ecossistema original os componentes adicionados

por este trabalho. Linux nativo e Docker com imagens base Linux são suportados nesta arquitetura.

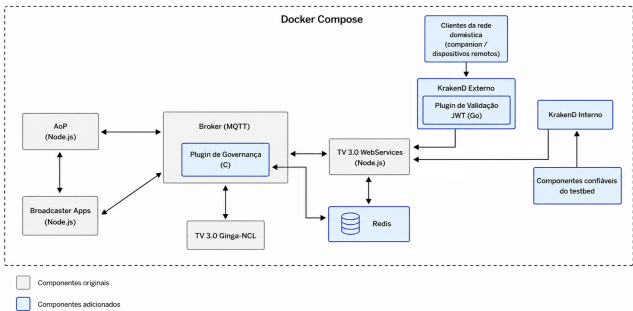


Figure 4: Arquitetura estendida do *testbed*. Os componentes em destaque foram adicionados neste trabalho: o *plugin* de governança embutido no *broker* Mosquitto, o *store* Redis e os dois *gateways* KrakenD, sendo o externo equipado com um *plugin* de validação de JWT em Go. O conjunto é orquestrado via Docker Compose. Adaptada de [9].

5 AVALIAÇÃO

A avaliação examina a contribuição central deste trabalho do ponto de vista da reprodutibilidade: o quanto a arquitetura containerizada reduz o esforço necessário para colocar o *testbed* em execução, em comparação com a arquitetura original. A análise é feita por inspeção do artefato, contrastando as dependências de sistema e os passos manuais exigidos por cada abordagem.

5.1 Esforço de Configuração do Ambiente

A Tabela 3 resume a comparação. A arquitetura original exige a instalação manual de cinco dependências de sistema (Mosquitto, Node.js, PM2, NW.js e FFmpeg) e mantém quatro arquivos de configuração específicos por sistema operacional, com a execução orquestrada externamente pelo PM2. A versão containerizada reduz a dependência de *host* a uma única (Docker), substitui os arquivos por sistema operacional por perfis do Compose e provisiona os serviços com um único comando em ambiente Linux.

Aspecto	Original	Containerizada
Dependências de <i>host</i>	5 (manuais) Mosquitto, Node.js, PM2, NW.js, FFmpeg	1 (Docker)
Config. por SO	4 arquivos (linux/mac/ wsl/setexpo)	perfis do Compose
Provisionamento	pm2 start	docker compose up
Comandos manuais	múltiplos	um (Linux)

Table 3: Comparação da quantidade de elementos instalados manualmente entre a arquitetura original e a containerizada.

A redução é mais relevante do que a contagem sugere. Entre as dependências eliminadas estão NW.js e FFmpeg, sensíveis a versão de sistema operacional e historicamente fonte de conflitos de *build*

na reprodução do ambiente. Ao encapsulá-las em imagens com versões fixas, a abordagem containerizada torna o provisionamento reprodutível entre diferentes máquinas e grupos de pesquisa. Para verificar essa reprodutibilidade na prática, o provisionamento foi executado em máquinas com configurações distintas de *hardware* e sistema operacional, incluindo Ubuntu 22.04, 24.04, 26.04 e Fedora 40, com Docker Engine na sua versão mais recente.

5.2 Verificação Funcional do Validador

O validador de esquemas foi submetido a testes de sucesso e de falha para cada operador da Tabela 2, cobrindo os tipos de dado do JSON Schema: booleanos, inteiros, números, *strings*, *arrays* e objetos, incluindo estruturas aninhadas. Os casos de sucesso utilizam *payloads* conformes ao esquema cadastrado e verificam que a publicação é aceita; os casos de falha violam individualmente cada operador (e.g., tipo incorreto, valor fora do intervalo, campo obrigatório ausente) e verificam que a publicação é rejeitada, com o erro correspondente registrado no *log* do *broker*. Todos os casos apresentaram o comportamento esperado.

5.3 Latência com o *Plugin* de Governança

Para quantificar o custo introduzido pelo *plugin* no caminho de mensagens, mediu-se a latência do ciclo completo de requisição e resposta sobre o *broker* MQTT: um cliente publica uma mensagem, que é validada pelo *plugin* e retida pelo *broker*; um segundo cliente a consome e publica a resposta, que passa novamente pela validação e retenção antes de ser entregue ao cliente original. Cada medição compreende, portanto, duas publicações e duas passagens pelo *plugin*, incluindo a validação de esquema dos *payloads*. Cada configuração foi executada em 5 repetições, variando o número de clientes simultâneos conectados ao *broker*. A Tabela 4 apresenta a latência média do ciclo por número de clientes.

Clientes	Latência média (ms)
10	1,37
20	1,58
50	2,74
100	3,53

Table 4: Latência média do ciclo de requisição e resposta via *broker* MQTT, com duas passagens pela validação do *plugin* de governança, por número de clientes simultâneos (5 repetições por configuração).

A latência cresce de forma moderada com a concorrência, de 1,37 ms com 10 clientes a 3,53 ms com 100 clientes — o que corresponde a menos de 2 ms por publicação validada mesmo na maior concorrência avaliada; no pior caso observado, o percentil 95 do ciclo foi de 7 ms, sem perda de mensagens em nenhuma configuração. Esses valores são compatíveis com os limiares relatados para o *testbed* original [10], indicando que a governança adicionada não compromete o uso previsto da plataforma.

5.4 Extensibilidade sem Acoplamento

As extensões propostas foram integradas sem modificação dos componentes existentes do *testbed*. Cada contribuição foi adicionada

como um novo elemento, o *plugin* de governança no *broker*, os *gateways* de API e o *microfrontend* de criação de perfis, preservando a interface dos componentes originais. Isso confirma, na prática, a propriedade de extensibilidade que motiva o uso do *testbed* como base para desenvolvimento colaborativo.

6 DISCUSSÃO E LIMITAÇÕES

Os resultados indicam que a containerização atende ao objetivo de reprodutibilidade do *testbed*, reduzindo o esforço de configuração a uma dependência única e a um provisionamento por comando único. Esta seção discute o alcance desse resultado e registra as limitações da abordagem e da avaliação.

A primeira limitação refere-se à política de validação de esquemas. A adoção de *allow-by-default* para tópicos sem esquema cadastrado favorece a adoção incremental, mas implica que a cobertura da validação depende do conjunto de esquemas registrados: tópicos ainda não cobertos trafegam sem verificação. A ampliação dessa cobertura para todos os tópicos definidos na ABNT NBR 25608 permanece como trabalho em andamento.

A segunda limitação diz respeito ao método de avaliação. A redução de esforço de configuração foi avaliada por inspeção do artefato, contrastando dependências e passos de provisionamento entre as duas arquiteturas. Essa análise evidencia a diferença estrutural entre as abordagens, mas não quantifica o ganho percebido por quem utiliza a plataforma. Uma avaliação com desenvolvedores, que meça tempo de configuração e esforço percebido em diferentes perfis de usuário, constitui um passo necessário para corroborar o benefício relatado e está prevista como desdobramento futuro.

7 CONCLUSÃO

Este trabalho demonstrou que o *testbed* distribuído da TV 3.0 AoP pode ser estendido em cada uma de suas três camadas, ao mesmo tempo em que fechou lacunas de governança de dados e reprodutibilidade da arquitetura original [9, 10]. Na camada de *back-end*, um modelo de governança em tempo de execução integrado ao *broker* MQTT combina controle de acesso por cliente, verificação de consentimento em conformidade com a ABNT NBR 25608 e a ISO/IEC 27560, e validação de esquema JSON dos *payloads*. Na camada de *WebServices*, dois *gateways* de API centralizam a autenticação e a verificação de consentimento para clientes externos e internos. Na camada visual, um novo *microfrontend* de criação de perfis demonstrou que a interface pode ser estendida sem modificação dos componentes existentes. Todo o ambiente é containerizado via Docker Compose, tornando-o reproduzível com um único comando em ambientes Linux.

A principal contribuição deste trabalho é demonstrar que o *testbed* da TV 3.0 AoP é extensível em todas as suas camadas sobre um ambiente comum de microserviços, troca de dados via MQTT e persistência em Redis. Cada extensão apresentada foi integrada sem acoplamento com os componentes existentes, validando o ambiente como base para o desenvolvimento colaborativo da plataforma. O fato de a validação, o controle de acesso e o consentimento serem implementados em um único *plugin* nativo, sem substituir o *broker* nem introduzir componentes externos ao caminho de mensagens, mantém a arquitetura de microserviços e o modelo de

estado retido propostos em [9], apenas adicionando governança antes da retenção.

Como desdobramentos futuros, pretende-se estender os esquemas JSON cadastrados para cobrir todos os tópicos definidos na ABNT NBR 25608. Também se pretende avaliar o impacto da validação em cenários de alta frequência de publicação, como atualizações de estado de nós Ginga-NCL durante apresentações interativas. Pretende-se também conduzir um estudo de usabilidade com desenvolvedores da plataforma para quantificar o ganho de esforço de configuração.

CUIDADOS ÉTICOS

Não se aplica. Este trabalho não envolveu pesquisa com seres humanos nem coleta, uso ou análise de dados pessoais. Os perfis de telespectador utilizados são fictícios, criados apenas para fins de demonstração da plataforma.

DECLARAÇÃO SOBRE O USO DE IA

Na revisão e no aprimoramento da redação deste artigo foram utilizadas ferramentas de Inteligência Artificial Generativa como apoio. A concepção, a implementação, os resultados e as conclusões do trabalho são de responsabilidade integral das pessoas autoras.

REFERENCES

- [1] 2023. ISO/IEC 27560:2023 – Privacy technologies – Consent record information structure. International Standard. International Organization for Standardization, Geneva, Switzerland.
- [2] Bestoun S Ahmed and Miroslav Bures. 2018. Testing of Smart TV Applications: Key Ingredients, Challenges and Proposed Solutions. In *Proceedings of the Future Technologies Conference*. Springer, 241–256.
- [3] Bestoun S Ahmed, Angelo Gargantini, and Miroslav Bures. 2020. An Automated Testing Framework for Smart TV Apps Based on Model Separation. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 62–73.
- [4] Associação Brasileira de Normas Técnicas. 2023. *Televisão digital terrestre – Codificação de dados e especificações de transmissão para radiofusão digital – Parte 2: Ginga-NCL para receptores fixos e móveis – Linguagem de aplicação XML para codificação de aplicações*. Associação Brasileira de Normas Técnicas.
- [5] Associação Brasileira de Normas Técnicas. 2025. ABNT NBR 25608: TV 3.0 – Application Coding.
- [6] Fábio Barreto, Raphael S. de Abreu, Marina I. P. Josué, Eyre Brasil B. Montevecchi, Pedro Alves Valentim, and Débora C. Muchaluat-Saade. 2023. Providing multimodal and multi-user interactions for digital TV applications. *Multimedia Tools and Applications* 82 (2023), 4821–4846.
- [7] Andrya K. O. Da Silva, Luzia C. Amorim, Orlewilson B. Maia, Eddie B. De Lima Filho, Tássia C. Da Rocha, Gabriel C. Dos Santos, Adrian L. Guedes, Andreza R. Queiroz, and Kawê R. C. De Araújo. 2024. An Automated Testing Methodology to Evaluate Multimedia Processing Stacks Running on Digital TV Receivers. In *2024 IEEE International Conference on Consumer Electronics (ICCE)*. 1–6.
- [8] Docker Inc. 2024. Docker: Accelerated Container Application Development. <https://www.docker.com>. Acessado em: abr. 2026.
- [9] Joel dos Santos, Rodrigo da Mota Sodré, Luís Claudio Ribeiro Junior, Glauco Amorim, and Débora C. Muchaluat-Saade. 2025. Uma Arquitetura para Experimentação de Novos Serviços na Plataforma Orientada a Aplicações da TV 3.0. In *Anais Estendidos do XXXI Simpósio Brasileiro de Sistemas Multimídia e Web (WebMedia 2025) – VIII Workshop Futuro da TV Digital Interativa (WTVDI 2025)*. Brazilian Computer Society, Rio de Janeiro, RJ, Brasil.
- [10] Joel A. F. dos Santos, Rodrigo M. Sodré, Pedro J. G. Castro, Luís C. R. Ribeiro Junior, Glauco F. Amorim, and Débora C. Muchaluat-Saade. 2025. An Open Testbed for the TV 3.0 Application Coding Layer. *IEEE Transactions on Broadcasting* (2025). Submitted November 2025.
- [11] EMQ Technologies. 2024. EMQX: The World's Most Scalable MQTT Broker. <https://www.emqx.io>. Acessado em: abr. 2026.
- [12] Michael Geers. 2020. *Micro Frontends in Action*. Manning Publications, Shelter Island, NY, USA.
- [13] Marina Josué, Marcelo F. Moreno, and Débora C. Muchaluat-Saade. 2024. Automatic Preparation of Sensory Effects: Managing Synchronization in Multimedial

- Applications. In *Proceedings of the 15th ACM Multimedia Systems Conference*. 100–108.
- [14] JSON Schema Organization. 2020. JSON Schema Specification – Draft 2020-12. <https://json-schema.org/specification>. Acessado em: ago. 2026.
- [15] Lura Project. 2024. KrakenD: Open Source API Gateway. <https://www.krakend.io>. Acessado em: abr. 2026.
- [16] Orlewilson B. Maia, Andre R. da Silva Conceição, Manoel J. de Souza Júnior, Fabrício Izumi, Eddie B. de Lima Filho, and Paulo Corrêa. 2022. A Real-Time Analyzer for Testing DTV Play. In *2022 IEEE International Conference on Consumer Electronics (ICCE)*. 01–05.
- [17] Marcelo F. Moreno, Débora Muchaluat-Saade, Guido Lemos, Sérgio Colcher, Carlos Soares Neto, Li-Chang Shuen C. S. Sousa, and Joel dos Santos. 2024. *TV 3.0: Especificações da Camada de Codificação de Aplicações*. Sociedade Brasileira de Computação, Chapter 4.
- [18] Redis Ltd. 2024. Redis: The Real-Time Data Platform. <https://redis.io>. Acessado em: abr. 2026.
- [19] Gabriel Souza, Daniel Silva, Matheus Delgado, Renato Rodrigues, Paulo R. C. Mendes, Glauco Fiorott Amorim, Alan L. V. Guedes, and Joel dos Santos. 2020. Interactive 360-degree Videos in Ginga-NCL Using Head-Mounted-Displays as Second Screen Devices. In *Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia '20)*. Association for Computing Machinery, New York, NY, USA, 289–296. doi:10.1145/3428658.3430972