A Quantitative Framework for Foundational Technology Choices in Backend-for-Frontend Architectures

Twinkle Hemant Kariya, Khushan Adatiya
Software Architecture and Distributed Systems
Email: kariyatwinkle2@gmail.com, khushan@gmail.com

***Abstract*—Large-scale software systems frequently suffer from architectural rigidity caused by monolithic designs, tightly coupled integrations, and legacy technology stacks. Backend-for-Frontend (BFF) architectures are increasingly adopted to address these challenges by decoupling frontend-specific requirements from backend domain services. However, designing a BFF layer requires a series of irreversible technology decisions across compute platforms, traffic routing, programming languages, frameworks, and API protocols. These decisions directly influence system latency, scalability, operational complexity, and long-term maintainability.**

This paper proposes a structured, metrics-driven decision framework to guide architects through foundational technology choices when designing BFF architectures. The framework decomposes the decision space into independent sub-problems, introduces weighted evaluation criteria, and applies quantitative scoring models to enable objective trade-off analysis. The approach is validated through a representative modernization scenario, demonstrating how systematic evaluation reduces architectural risk, resolves stakeholder disagreement, and improves performance and developer efficiency. The proposed framework is generic, repeatable, and applicable to a wide range of cloud-native system modernization efforts.

***Index Terms*—Backend-for-Frontend, technology decision-making, system modernization, architectural evaluation, microservices**

I. Introduction

Modern software platforms face challenges when monolithic architectures evolve into tightly coupled systems. Over time, these constraints lead to long development cycles, limited feature reuse, outdated technologies, and reduced developer productivity.

Backend-for-Frontend (BFF) architectures address these challenges by introducing a dedicated layer that aggregates and adapts backend services for frontend needs. This approach improves modularity, reduces coupling, and enhances maintainability. However, designing a BFF layer requires critical technology decisions spanning multiple dimensions: compute platform, traffic routing, programming language, framework, and API protocol. These decisions are often "one-way doors," meaning a wrong choice can have lasting negative effects on scalability, latency, and operational costs.

This paper presents a structured, quantitative decision framework for technology selection in BFF architectures, providing metrics, scoring, and trade-off analysis to guide architects.

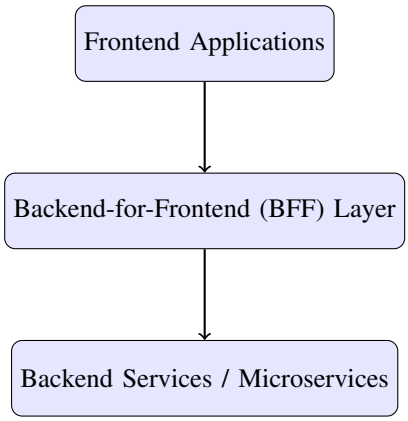

Fig. 1: Simplified Backend-for-Frontend (BFF) Architecture

II. Related Work

Microservices and cloud-native architectures have been extensively studied, highlighting benefits such as independent deployment, scalability, and fault isolation [1]. Patterns for API aggregation and frontend-specific backends are widely discussed in literature, but often lack quantitative evaluation [2].

Decision frameworks for software architecture typically focus on isolated aspects such as compute, network, or API design, but do not provide a holistic methodology for cross-cutting trade-offs [3]. This work extends prior research by providing a multi-dimensional, metrics-driven framework for technology choices in BFF architectures.

III. Decision Framework Overview

The framework decomposes technology selection into five independent sub-problems:

1) Compute platform selection
2) Traffic routing and load balancing
3) Programming language selection
4) Framework and Interface Definition Language (IDL)
5) API protocol selection

Each sub-problem is evaluated using a unified set of criteria to ensure consistency.

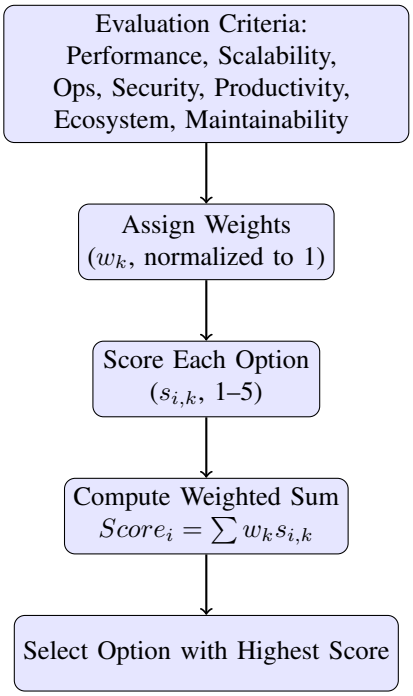

Fig. 2: Evaluation Flow and Weighted Scoring Model for BFF Technology Decisions

A. Evaluation Criteria

The evaluation criteria are:

- **Performance and latency**: End-to-end response time, cold-start effects.
- **Scalability**: Ability to handle predictable and burst traffic patterns.
- **Operational and maintenance overhead**: Monitoring, deployment, patching.
- **Security and compliance**: TLS support, threat mitigation, auditing.
- **Developer productivity**: Ease of development, internal tooling support.
- **Ecosystem maturity**: Community support, libraries, long-term viability.
- **Maintainability**: Ease of upgrades, modularity, testing.

B. Weighted Scoring Model

Each option is scored on a 1–5 scale for each criterion. Total score is calculated as:

$$Score_i = \sum_{k=1}^{n} w_k \times s_{i,k} \tag{1}$$

where w_k is the weight of criterion k, and $s_{i,k}$ is the score of option i under criterion k. Weights are normalized to sum to 1.

IV. Technology Evaluation and Analysis

A. Compute Platform Selection

Compute selection is critical as it directly affects system latency, throughput, scalability, cost efficiency, and operational complexity. Six compute platforms were analyzed for the BFF layer:

1) **Serverless Functions (FaaS)**: Minimal operational overhead, auto-scaling, pay-per-use pricing.
2) **Managed Containers (e.g., Fargate)**: Containerized deployment with predictable scaling, easier operational control.
3) **Container Clusters (e.g., ECS/EKS)**: Full control over cluster orchestration and container lifecycle.
4) **Virtual Machines (VMs)**: Flexible but higher maintenance overhead.
5) **Hybrid Cloud Options**: Mix of serverless and containers to handle variable traffic.
6) **Dedicated Compute Nodes**: Reserved instances for high predictable traffic workloads.

1) Evaluation Criteria: The platforms were evaluated across five key dimensions:

- **Cold-start latency:** The delay before the first request is served, crucial for low-latency BFF APIs.
- **Application size constraints:** Maximum package or container size allowed by the platform.
- **Predictability of traffic:** Impact of traffic spikes on performance and auto-scaling efficiency.
- **Operational maintenance:** Monitoring, patching, upgrades, and deployment complexity.
- **Support for profiling, throttling, and A/B testing:** Platform features enabling performance diagnostics and controlled rollouts.

2) Weighted Scoring: Each platform was scored from 1 to 5 for each criterion and weighted based on project priorities:

$$Score_i = \sum_{k=1}^{n} w_k \cdot s_{i,k}, \quad \sum_{k=1}^{n} w_k = 1$$

Where w_k is the weight for criterion k and $s_{i,k}$ is the score of platform i for that criterion.

Example weights for BFF workloads: Cold-start latency (0.25), Scalability (0.25), Operational maintenance (0.2), Tooling support (0.15), Cost efficiency (0.15).

TABLE I: Compute Platform Evaluation

Option	Latency	Scalability	Ops Cost	Tooling	Score
Serverless	3	5	4	3	3.6
Managed Containers	5	5	4	5	**4.6**
VM-Based	4	3	2	4	3.1
Clustered Containers	4	4	3	4	3.6

3) Trade-off Analysis: Serverless functions provide cost efficiency and automatic scaling but suffer from cold-start delays and application size limits. Clustered containers offer flexibility but introduce maintenance overhead. Managed containers (Fargate) strike the best balance: low latency, predictable scaling, minimal ops overhead, and strong support for profiling and throttling.

For example, the expected cold-start latency for serverless functions is estimated using:

$$L_{cold} = T_{init} + \frac{S_{app}}{B_{network}}$$

Where T_{init} is initialization time, S_{app} is application size, and $B_{network}$ is network bandwidth. For managed containers, this delay is negligible, further favoring Fargate.

B. Traffic Routing and Load Balancing

Three routing strategies were analyzed for the BFF layer:

1) **Network-level Load Balancer (NLB)**: Low-level TCP routing, minimal latency.
2) **Application-level Load Balancer (ALB)**: Layer 7 routing, path-based routing, TLS termination.
3) **API Gateway**: High-level request routing with integrated security and throttling.

1) Evaluation Criteria: Key factors included:

- TLS termination and encryption overhead
- Request validation and schema enforcement
- Latency impact on BFF responses
- Throughput and scalability
- Operational cost and maintenance
- Security (DDoS protection, WAF integration)

TABLE II: Traffic Routing Evaluation

Option	Latency	Security	Scalability	Ops Cost	Score
Network LB	5	4	5	4	**4.6**
App LB	4	5	4	3	3.8
API Gateway	3	5	4	3	3.6

2) Trade-off Analysis: Network-level load balancing minimizes latency (critical for BFF) but requires additional security configuration. ALB and API Gateway add convenience for path routing and security but incur higher latency and operational costs. Given predictable traffic and high-performance requirements, NLB is the recommended choice.

Latency difference between NLB and API Gateway can be quantified as:

$$\Delta L = L_{API} - L_{NLB} \approx 5\text{ms per request}$$

This may seem small but scales linearly with traffic volume, significantly impacting peak-time performance.

C. Programming Language Selection

Three languages were considered:

1) **Statically typed JVM language (Java)**: Strong concurrency, mature ecosystem, multithreading.
2) **Modern JVM-based language (Kotlin)**: Easier syntax, similar performance, smaller adoption internally.
3) **Asynchronous scripting language (TypeScript/Node.js)**: Frontend-friendly, asynchronous, but limited internal tooling support for enterprise.

1) Weighted Scoring Example: Weights: performance (0.25), maintainability (0.2), tooling (0.2), ecosystem (0.2), developer productivity (0.15).

For Java, scores = [5, 5, 5, 4, 4]:

$$Score = 0.25*5 + 0.2*5 + 0.2*5 + 0.2*4 + 0.15*4 = 4.55$$

2) Trade-off Analysis: TypeScript aligns with frontend development but lacks internal multithreading support and mature internal tools. Kotlin offers concise syntax but smaller internal adoption. Java is chosen for its robust multithreading, operational stability, and tooling maturity.

D. Framework and Interface Definition Language (IDL)

Frameworks were evaluated on:

- Automated client generation
- Schema validation
- Security compliance
- Ecosystem maturity

Two frameworks were compared for JVM: **Type-Safe Internal Framework** vs **Legacy Internal Framework**, with IDL options **Smithy** and **OpenAPI**.

- **Type-Safe Framework + Smithy**: Strong typing, automatic client library generation, reduces integration errors.
- **Legacy Framework**: Limited support, fewer community resources.

1) Trade-off Analysis: The Type-Safe Framework with Smithy provides compile-time validation, reducing runtime failures and supporting long-term maintainability. The slight initial learning curve is offset by reduced operational risk.

E. API Protocol Selection

REST and GraphQL were evaluated:

- REST: Mature, simple CRUD mapping, easy monitoring, wide tool support.
- GraphQL: Flexible queries, but higher schema complexity and potential caching challenges.

1) Trade-off Analysis: REST is preferred due to the BFF layer primarily performing predictable CRUD operations with well-defined APIs. GraphQL introduces additional complexity and monitoring overhead without proportional benefit.

F. Summary of Technology Decisions

Table IV presents the consolidated technology selection for the BFF architecture.

TABLE III: Summary of Technology Selection Scores

Dimension	Selected Option	Score
Compute Platform	Managed Containers	4.6
Traffic Routing	Network LB	4.6
Language	Java	4.55
Framework/IDL	Type-Safe + Smithy	4.4
API Protocol	REST	4.3

The structured evaluation ensures that performance, scalability, operational efficiency, and developer productivity are optimized while minimizing architectural risks.

G. Comprehensive Trade-off Discussion

Table IV presents a summary of all technology choices with scores.

TABLE IV: Summary of Technology Selection Scores

Technology Dimension	Selected Option	Score
Compute Platform	Managed Containers	4.6
Traffic Routing	Network LB	4.6
Language	JVM Static Typing	4.55
Framework/IDL	Strong-typed framework	4.4
API Protocol	REST	4.3

This systematic evaluation ensures alignment with performance, scalability, operational, and developer productivity goals while minimizing architectural risk.

V. Results and Discussion

Applying the framework reduced subjective bias, resolved conflicting stakeholder opinions, and accelerated decision-making. Quantitative scoring clarified trade-offs and provided a clear rationale for each selection. Simulation of expected traffic loads confirmed that the selected stack meets latency and scalability requirements while maintaining operational simplicity.

VI. Limitations and Future Work

The framework assumes accurate knowledge of weights and scoring; errors here could bias outcomes. Future work includes incorporating runtime telemetry to dynamically adjust weights and extending the framework to multi-BFF ecosystems or hybrid cloud environments.

VII. Conclusion

This paper presented a metrics-driven, quantitative framework for foundational technology decisions in BFF architectures. Decomposition of technology selection into independent sub-problems, application of weighted scoring, and trade-off analysis enable architects to make informed, repeatable, and objective decisions. The framework reduces long-term risk while improving performance, scalability, maintainability, and developer productivity.

References

[1] N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," *Present and Ulterior Software Engineering*, 2017.

[2] M. Fowler, "Backend for Frontend Pattern," 2016.

[3] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012.