# Resonance: Replacing Software Constants with Context-Aware Models in Real-time Communication

**Jayant Gupchup, Ashkan Aazami, Yaran Fan, Senja Filipi, Tom Finley, Scott Inglis, Marcus Asteborg, Luke Caroll, Rajan Chari, Markus Cozowicz, Vishak Gopal, Vinod Prakash, Sasikanth Bendapudi, Jack Gerrits, Eric Lau, Huazhou Liu*, Marco Rossi, Dima Slobodianyk, Dmitri Birjukov, Matty Cooper, Nilesh Javar, Dmitriy Perednya, Sriram Srinivasan, John Langford, Ross Cutler, Johannes Gehrke**

Microsoft

## Abstract

Large software systems tune hundreds of 'constants' to optimize their runtime performance. These values are commonly derived through intuition, lab tests, or A/B tests. A 'one-size-fits-all' approach is often sub-optimal as the best value depends on runtime context. In this paper, we provide an experimental approach to replace constants with learned contextual functions for *Skype*- a widely used real-time communication (RTC) application. We present *Resonance*, a system based on contextual bandits (CB). We describe experiences from three real-world experiments: applying it to the audio, video, and transport components in *Skype*. We surface a unique and practical challenge of performing machine learning (ML) inference in large software systems written using encapsulation principles. Finally, we open-source *FeatureBroker*, a library to reduce the friction in adopting ML models in such development environments.

## 1 Introduction

Hyperparameter tuning of ML models using automation has a very rich literature [9, 2, 18, 8, 10, 17]. However, tuning of application constants such as buffer sizes, thresholds, or timeout settings is commonly done manually. Tuning these constants in live production systems requires overcoming many practical hurdles. In this paper, we study these challenges for a widely used RTC application.

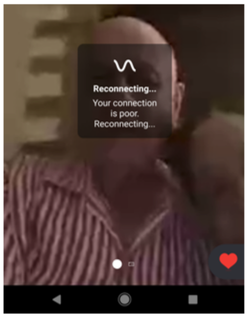
**Motivating Example**: Consider a video conference call in Skype [2] where one of the participants is using an unstable network. In Skype, if the endpoint stops receiving media packets, the application attempts to "reconnect" the call on an alternate interface (e.g., WiFi interface stops working, but 4G network is active). The reconnect threshold constant represents a careful trade-off. If the switch is triggered too early, the user's connection is unnecessarily moved into the "call reconnecting" status and starts to renegotiate all call parameters, leading to a false positive (a disruptive experience). If the application waits too long, the user loses patience and hangs up due to frustration. Furthermore, we found mobile users tend to hang up more quickly than desktop users, indicating the contextual nature of this problem as shown in Figure 1.

**Literature**: Recently, a few solutions and systems have emerged to address some of these challenges. Carbine et al. present *SmartChoices*, an API to replace heuristics in traditional algorithms with ML models built using standard reinforcement learning (RL) methods [5]. SmartChoices demonstrated the effectiveness on search, sort (pivot index), and cache algorithms. There are examples of applying RL-based methods in the query optimization and job scheduling domains, respectively [14, 11]. To replace runtime constants, we employ a simpler approach of CB as we found

*Now with Outreach. Work performed while at Microsoft.

[2]The ideas in this paper are applicable and generalize to the Microsoft Teams application too.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.



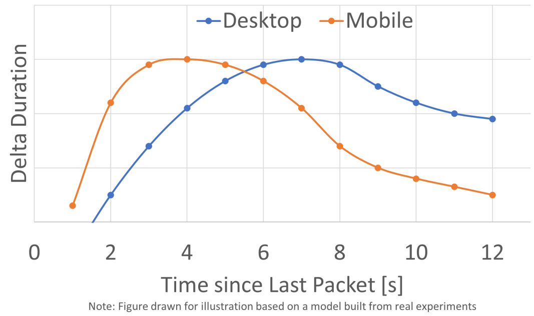




Figure 1: The left figure shows the *Network Reconnect* feature in Skype. The right figure shows the impact of trigger threshold on average call duration (ACD). [3]

it to scale effectively in our production scenarios compared to RL. By definition, a system deployed with a fixed constant does not capture any data on its performance for different values of that constant. The explore/exploit approach of CB methods offer a simple way to collect system response for different values of the constant, and learn a model to replace a constant (or a heuristic) based on application context [4, 3, 7]. Furthermore, a CB-based approach has the advantage of handing population drift as a certain percentage of traffic is reserved for exploration. Bakshy et al. describe *Ax*, a CB-based optimization and experimentation system for learning parameters at Facebook [3]. In *Resonance*, we address a novel and practical challenge of ML inference in real-world systems when context information is trapped behind API surfaces. Addressing this challenge in a production system sets our work apart from previous work applying CB models for tuning heuristics.

**Inference Challenges**: Large production software systems comprise of components interacting via strict API boundaries (or contracts). This encapsulation creates isolation, making it difficult to surface context (or features) to be consumed by other components for runtime inference (detailed example in Section 2). A naive approach requires several layers of API changes and alignment that result in the breaking of established contracts. We find that these high development costs were a big driver for the lack of adoption (or integration) of CB models relying on distributed context. It is noteworthy that this challenge is not unique to CB models, but rather any model requiring access to features distributed in siloed components. To scale adoption, we created a library called *FeatureBroker* that was able to address these challenges and bring down integration costs from 2 months to 2 days. It is noteworthy that the challenges addressed by *FeatureBroker* are applicable to many software systems, particularly large codebases that ship with 1000s of system constants.

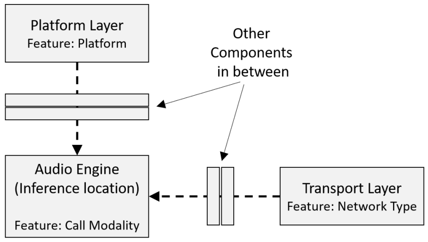
**Contributions**: This paper makes the following contributions:

- We outline *Resonance*, an experimental system to replace constants with CB models for RTC scenarios.
- We detail *FeatureBroker*, a library for managing the conversation between an inference engine and distributed features produced by components. We make this library open source for the benefit of the community [15].
- We present results from applying *FeatureBroker* to optimize Skype's audio, video, and transport components in production (Section 3).

## 2 Resonance

**System Overview**: *Resonance* comprises of the following components: Skype App, A/B test system, Client Inference (*CI*) and Learning Service (*LS*) as described in [1]. The benefit of replacing an *App* constant is evaluated as an A/B test; the control variant represents the fixed constant, whereas the treatment variant is represented by a CB model built using *VowpalWabbit (VW)* – a well known open source library for CB models [12]. The *CI* component performs runtime inference based on the App context as specified by the model. The *CI* component captures training logs comprising of {context value, recommended action, action probability, reward metric} and transmits them to the *LS*. The *LS* uses these training logs to produce a *VW* models (or policies) that get delivered to the client endpoints periodically (e.g., every 4 hours) completing the inference-log-learn loop.

[3]Figure drawn for illustration by modeling data gathered from real telemetry.




```
// Inference without FeatureBroker
jbHysteresis = model.predict([
    FeatureStore.GetValue("nwType"),
    FeatureStore.GetValue("platform"),
    FeatureStore.GetValue("callType")]);
```

Figure 2: Left figure shows the organization of components within Skype's RTC stack; illustrating the distributed nature of features in a well-encapsulated system. The right figure shows the inference performed in the absence of a library like *FeatureBroker*.

In this paper, we focus our attention on the novel *FeatureBroker* client library. In our explanations, we define context as a collection of feature values (e.g., network type = WiFi, platform = mobile).

**Feature Broker**: We motivate the need for such a library by way of an example. Consider we want to replace a constant *jbHysteris* in the audio component with the output of a CB model. A simple API to do that is shown in Figure 2 (right side).

While this API looks simple, practical problems emerge when performing inference using this approach. In this scenario, the context depends on the global state of the system, such as the platform (e.g., desktop/wired), the network type (e.g., wired/WiFi), and call modality (e.g., audio/video). Figure 2 (left side) shows the logical organization of components within our RTC application. While call modality information is available within the audio component, platform and network type information were produced in different components. Moreover, the audio and platform/transport components were separated by many other components and API surfaces. This situation where the features of interest are encapsulated in well-separated components was found to be a common pattern. Component owners want well-defined, stable public API surfaces for maintenance reasons. In addition, expanding them on a feature-by-feature basis is onerous. Furthermore, the components producing the features may not be running in the same thread as the component performing inference, and lastly, the changes in feature values at runtime (e.g., a mid-call change in network type) would require updating the prediction gracefully.

The idea of a simple shared thread-safe key-value feature store proved insufficient for two primary reasons:

1. *Coherence*: Inference libraries need feature values to be stable and consistent. However, as seen above, feature values can update asynchronously (e.g., network type can change mid-call).
2. *Hierarchy*: Features may be consumed by multiple models and inference components. For example, three models (*JBHysteresis*, *ScreenshareEncoding* and *NetworkReconnect*) may consume platform and network type features while having their own local features (e.g., call modality). The most graceful way to handle this was to make the structure hierarchical and allow it to be "forked" for sub-components.

The shared structure responsible for handling these problems is termed *FeatureBroker*. It manages the conversation between the client code that provides features (inputs) and consumes inference results (outputs) from a CB model. In this paper, it would be impossible to describe the library in detail. Instead, we highlight the main elements of the solution. Interested readers can see a detailed real-world example in the open-source version of the library *FeatureBroker* [15]. The key operations of the *FeatureBroker* are:

1. *Binding Inputs*: Given a name and type returns an "input pipe" into which a feature providing component can feed values.
2. *Associating Models*: Register and describe a scheme for transforming inputs to outputs.
3. *Binding Outputs*: Returns an "output pipe" in which inference consuming models can query for updated values.

The notion of binding draws parallels to binding values for form elements obtained from a database [13]. Finally, it is noteworthy that *FeatureBroker* can be used with any inference library (e.g., ONNX [16]) and this concept generalizes to any inference scenario where features are distributed across components. Detailed examples can be found online [15].

| Experiment Name | *Audio Jitter Buffer* | *Screen Share Encoding* | *Network Reconnect* |
|---|---|---|---|
| Platforms | Desktop | {Desktop, Mobile} | {Desktop, Mobile} |
| Impressions (Millions) | 4M | 7.6M | 19M |
| RTC Component | Audio | Video | Transport |
| Number of Actions | 10 | 10 | 5 |
| Unique Contexts | 14 | 440 | 390 |
| Metric | Poor-Audio-rating | Poor-Video-rating | Call Duration (CD) |
| Model Size | 4 KB | 22 KB | 10 KB |
| Metric Improvement (%) | 1.1% | 9.9% | 4.2% |

Table 1: The table captures the overall Summary results from experiments conducted on the representative scenarios.

## 3 Experimental Results

We present results from three scenarios in Skype. The scenario related to learning the reconnect threshold was described in Section 1. We introduce two other scenarios.

**Scenario: Audio Jitter Buffer (JB) Hysteresis** The JB component absorbs variability in network packet arrivals (i.e., jitter) to present a smooth playout of audio frames transmitted via the network. *JBHysteresis* is a meta-parameter controlling the amount of inertia associated with changes in the buffer size. The optimization metric to learn this parameter was *Poor-Audio-Rating*, an objective metric estimating speech quality based on packet loss, jitter, and conversational delay derived from user ratings [6].

**Scenario: Video Encoding Bitrate Allocation for Screen Share** Screen sharing sessions in VoIP require careful selection of the bitrate. Large bit rates lead to a higher image quality, but are also susceptible to frame freezes due to packet losses under poor network conditions. This quality-distortion tradeoff is characterized by a constant that weights transmission rates and frame freezes. Similar to audio, we optimized for a video technical metric termed *Poor-Video-Rating*.

**Experiments and Results**: A summary of the experimental data for all three scenarios is presented in Table 1. The baseline (control group) for these experiments is the metric value obtained when using the 'one-size-fit-all' constant value. The metric improvement represents the relative gain observed when replacing the constant with a context-aware model (treatment group). The data is collected using an $\epsilon$-greedy policy. The $\epsilon$ was set to 0.2 for the experiments. For each experiment, we report the number of unique contexts to convey the range of inputs going into the *FeatureBroker*. Each of these experiments showed statistically significant improvements with a p-value less than 0.01. We note the following:

- These experiments show that we could successfully replace constants with CB models for RTC scenarios. Since these experiments were done in three different components by three different teams, it demonstrates the generality of the *FeatureBroker* solution.
- New experiments were able to build and re-use the on-boarding of features done by previous experiments. For example, the on-boarding of features for *ScreenShareEncoding* experiment simplified the integration of the *NetworkReconnect* model as the two models shared features. Prior to the introduction of *FeatureBroker*, engineering teams would have to coordinate and align on API changes to expose features. This process would often take multiple days, and has been reduced by an order of magnitude.
- Due to the limited memory and CPU budget in RTC applications, we focused on keeping the footprint of CB models small. The main reason was to scale this methodology so that it can be applied to 100s of scenarios within Skype.

## 4 Discussion

This paper introduces *Resonance*, a system for replacing application constants with context-aware models. We built and deployed this system for the real-time production scenarios of Skype. We presented *FeatureBroker*, a library for integrating ML models relying on global context. The library presents a solution to the problem of inference when the context is distributed across strong component boundaries. We make this library available to practitioners to bring down the adoption and development costs. Using three statistically significant real-world experiments, we showed this methodology can improve system performance. Such a methodology provides a new tool for component owners to optimize their components.

**Future Work**: The problem of efficient discovery of sensitive constants in a live system containing thousands of constants remains challenging.

In conclusion, we emphasize that the ideas presented in *Resonance* and challenges solved by *FeatureBroker* are applicable to many software systems, particularly large codebases that ship with 1000s of system constants.

## References


[1] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, et al. A multiworld testing decision service. *arXiv preprint arXiv:1606.03966*, 7, 2016.

[2] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631. ACM, 2019.

[3] E. Bakshy. Towards simplicity in machine learning for live systems with adaptive experimentation. `https://slideslive.com/38922475/towards-simplicity-in-machine-learning-for-live-systems-with-adaptive-experimentation`, 2019.

[4] E. Bakshy, L. Dworkin, B. Karrer, K. Kashin, B. Letham, A. Murthy, and S. Singh. Ae: A domain-agnostic platform for adaptive experimentation. In *Systems for ML at NeurIPS*, 2018.

[5] V. Carbune, T. Coppey, A. Daryin, T. Deselaers, N. Sarda, and J. Yagnik. Smartchoices: Hybridizing programming and machine learning. In *Reinforcement Learning for Real Life (RL4RealLife) Workshop in the 36th International Conference on Machine Learning (ICML),*, 2019.

[6] L. Ding and R. A. Goubran. Speech quality prediction in voip using the extended e-model. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, volume 7, pages 3974–3978. IEEE, 2003.

[7] G. Dulac-Arnold, D. Mankowitz, and T. Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

[8] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[9] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1487–1495. ACM, 2017.

[10] H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1946–1956. ACM, 2019.

[11] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[12] J. Langford, L. Li, and A. Strehl. Vowpal wabbit online learning project. `https://github.com/VowpalWabbit/vowpal_wabbit/wiki`, 2007.

[13] J. Lerman. *Programming Entity Framework: Building Data Centric Apps with the ADO. NET Entity Framework*. " O'Reilly Media, Inc.", 2010.

[14] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[15] Microsoft. Feature broker, a library for enabling sharing of context/features across code components for ml inference. `https://github.com/microsoft/FeatureBroker`.

[16] ONNX Runtime: cross-platform, high performance scoring engine for ML models. `https://github.com/microsoft/onnxruntime`, 2019.

[17] S. Paul, V. Kurin, and S. Whiteson. Fast efficient hyperparameter tuning for policy gradient methods. In *Advances in Neural Information Processing Systems*, pages 4616–4626, 2019.

[18] V. Perrone, H. Shen, M. W. Seeger, C. Archambeau, and R. Jenatton. Learning search spaces for bayesian optimization: Another view of hyperparameter transfer learning. In *Advances in Neural Information Processing Systems*, pages 12771–12781, 2019.