MatchOpt: A Distributed Matchmaking Engine Based on EOMM

Kevin Lu Fang
MSCS, UC Irvine
Irvine, USA
klfang1 @ uci.edu

Andrew Joe Collins
MSCS, UC Irvine
Irvine, USA
collina2 @ uci.edu

Gnana Heemmanshuu Dasari
MSCS, UC Irvine
Irvine, USA
ghdasari @ uci.edu

***Abstract*—Matchmaking is essential to online multiplayer games, traditionally aiming to ensure fair matches using methods like Skill-Based Matchmaking (SBMM) or ELO. However, these approaches do not account for player retention. Engagement Optimized Matchmaking (EOMM) introduces a churn-aware framework that optimizes for long-term player engagement. Despite its potential, no scalable, open-source system currently supports EOMM, and existing platforms like AWS GameLift and PlayFab are proprietary and inflexible. OpenMatch, while open-source, is tightly coupled to SBMM, difficult to extend, and burdensome to deploy due to its reliance on gRPC, Redis, and Kubernetes.**

We introduce MatchOpt, a scalable, open-source matchmaking system designed specifically to support engagement-focused algorithms like EOMM. Built with Kafka for load balancing, Apache Flink for real-time processing, and Docker for modular deployment, MatchOpt is fault-tolerant and efficient. We evaluate it locally under controlled conditions, showing strong performance across latency, throughput, and player engagement metrics, even at increasing scale. Our results validate EOMM's benefits and demonstrate that MatchOpt can support complex matchmaking logic without sacrificing performance. Future work includes deployment to Kubernetes, autoscaling support, and abstraction layers for pluggable churn models and matchmaking algorithms.

***Keywords*—** Engagement-optimized matchmaking, video game matchmaking engine, distributed matchmaking

I. Introduction

Effective matchmaking is fundamental to the player experience in online games, as it directly impacts both perceived fairness and overall engagement. A matchmaking system is meant to match users with matches that offer good balance between latency, skill, and other features of the game. Traditional matchmaking systems will place a strong focus on trying to measure and match players in a way that maintains restricted skill variation. While traditional matchmaking is still effective and may still be necessary for a lot of gamers, in today's world where player engagement, monetization, and retention are critical for success, the drawbacks of only being skill-based are becoming evident.

There has been greater awareness on the role of player engagement in online games, but a scalable matchmaking system with Engagement-Optimized Matchmaking (EOMM) is not yet available commercially. Most matchmaking systems in production still use a conventional approach called skill-based matchmaking (SBMM), which prioritizes matchmaking players based on their skill level, and does not consider player retention or the potential for churn to occur. EOMM presents an exciting alternative, as its aim is to optimize long-term player engagement, but there are no existing open-source or commercial systems currently; or are flexible and scalable for practical application. The failure to develop infrastructure in the field of game backend technology that supports a native EOMM paradigm has led to an egregious central confusion in the video game industry.

To facilitate reproducibility and further development, the full implementation of MatchOpt is open-sourced and available at: https://github.com/heemmanshuu/thamm

II. Related Work

Multiple matchmaking algorithms have been proposed and implemented in both industry and academia, each with a unique trade-off of fairness, engagement, and complexity. Skill-Based Matchmaking (SBMM) aims to match players at their own skill levels (i.e., win rate, player rank, etc.). SBMM does emphasize fairness, but can lead to monotonous and predictable gameplay. ELO is a widely used rating system for 1v1 competitive games. ELO is named after the developer of the chess algorithm Arpad Elo and modifies the player ratings based on the results of the matches, updating their ratings to reflect the skill of the player over time [1].

EOMM is a more recent and fluid form of matchmaking. Rather than analyzing fairness, EOMM analyzes the best means to retain players by modeling players' likelihood of churn through machine learning, and matching players in a manner that is more likely to lead to long-term retention [2].

On the side of system design, OpenMatch is the only open source matchmaking framework in operation with matchmaking logic offered as a service. OpenMatch is primarily captured around more traditional skill-based systems. More exploratory algorithms are being researched, including GloMatch [3], which uses reinforcement learning to balance engagement fairness; EnMatch [4], which seeks entropy-based, improved fairness and stability; and CUPID [5] focusing on maximized latency and level of skill matching using clustering and prediction methodologies.

Of all of these algorithms, EOMM uniquely positions itself as the only method that explicitly seeks player retention as the optimization target, making it the ideal matchmaking approach for live-service games that prioritize long-term player engagement.

A. EOMM: A Matchmaking Model To Retain Players in the Game

Diving deeper into EOMM, first there is the Churn Prediction Model, which is a machine learning model that has been trained to estimate the probability that a player will stop playing (churn) after a particular match. Second, there is Outcome Simulation, which gives a prediction of what the likely outcome will be of a match between any two players or teams using historically and contextually relevant data. Lastly, there is Optimization Graph, where players are nodes and matches are edges, and the system actually solves for the various combinations of matches that achieve the lowest (predicted) churn per the entire pool of players. In the end, this approach allows EOMM to not just concern itself with fairness, but to directly optimize for engagement, which is particularly relevant in today's player-retention enhancing game environments.

By considering characteristics of each player like player behavior, recent performance, engagement history, and predicted outcome, EOMM seeks to match players to maximize potential enjoyment and player retention. This is why it is well-suited for live-service games since player longevity is a key success metric for the business.

EOMM is fundamentally an approach that shifts our design aims from fairness to engagement. Since there is potential for EOMM to impact player retention and lifetime value, we decided to implement it as the base of our matchmaking system. Using EOMM as a foundation allows us to create a matchmaking system with less focus on equating performance levels in an immediate tracking sense, but rather create a more satisfying experience for players for the long-term.

B. *Existing Matchmaking Systems*

Today, many matchmaking platforms exist, either as cloud-hosted services or as technologies including in-house functionality from major platforms. Microsoft Matchmaker is a part of the Azure PlayFab ecosystem, has limited open-source capabilities, and provides no positioning or extensibility of their internal algorithms for modification; it supports rule-based matchmaking. Unity Matchmaker provides matchmaking with your pre-defined rules and skill rating. But, it too does not expose any working context of transparency or extensibility. Idem Matchmaker provides a rules-based matching solution, as well, with some degrees of freedom but it does not develop towards engagement optimized objectives like EOMM. AWS GameLift FlexMatch allows developers to create their matchmaking rules on JSON-based schemas which allow flexibility to develop based on either SBMM or latency state-limited pairing: but again, it doesn't develop towards something verifiably advanced like data-driven optimization against EOMM. Microsoft PlayFab Matchmaking however has managed matchmaking capabilities without definitive tagging- or scoring features so similarly, it states with indistinct trajectories and troubling adjustments.

All of these systems share similar limitations, as they are all closed-source, non-extensible, and infrastructure opaque. Developers cannot access or modify the core matchmaking logic and they cannot easily understand the method for match decision-making. Even more significantly, none of the systems allow hooked EOMM nor provide the machine learning interfaces necessary to integrate churn prediction models and dynamic optimization graphs. This is why they are inadequate for engagement-optimized matchmaking at scale.

C. *OpenMatch: Limitations for EOMM*

OpenMatch is the only open-source matchmaking framework available today. Developed by Google Cloud and Unity, it provides a pluggable, scalable backend that enables developers to specify their own match function, and connect their match logic services using gRPC [6]. While OpenMatch provides useful primitives to work with match tickets, pools, and assignment workflows, it is primarily designed for skill-based matchmaking (SBMM). It was built with a many-to-many matching problem in mind and where core logic is expressed by relatively simple rules or functions.

However, OpenMatch has several significant limitations. It has no out-of-the-box functionality for Engagement Optimized Matchmaking (EOMM); it has no ability to include and implement machine learning models; it has no facility for churn prediction pipelines; and it cannot optimize non-responsiveness based on predicted match outcomes. Further to this, the infrastructure required to deploy OpenMatch is complex. It requires a full Kubernetes cluster and running Redis for state storage, and encompasses managing all gRPC based services and workers. This creates a high operational learning curve, and makes it difficult for small teams to experiment and iterate quickly.

Everything considered, OpenMatch is technically open-source, however, its architectural choices have limitations making it not possible to limitlessly expand its capabilities. Within this architectural solution, it would be very difficult to integrate advanced machine learning pipelines, and even simply adapting the matchmaking optimization logic would be a challenge. Therefore, due to the limitations of OpenMatch, and given its structure, it's not a practical launching pad to develop EOMM. The design is tied more to traditional matchmaking paradigms, and the lack of support for data-driven engagement optimization represents a considerable void for developers and researchers aiming to advance matchmaking innovations.

III. MatchOpt Architecture

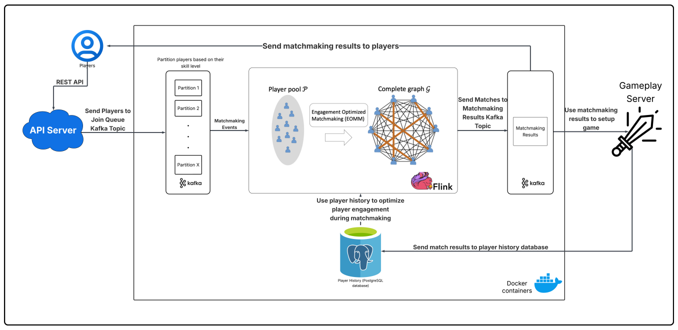
Send matchmaking results to players
Players
REST API
API Server
Send Players to Join Queue Kafka Topic
Partition players based on their skill level
Partition 1
Partition 2
Partition X
kafka
Matchmaking Events
Player pool $\mathcal{P}$
Engagement Optimized Matchmaking (EOMM)
Complete graph $\mathcal{G}$
Flink
Send Matches to Matchmaking Results Kafka Topic
Matchmaking Results
kafka
Use matchmaking results to setup game
Gameplay Server
Use player history to optimize player engagement during matchmaking
Player History (PostgreSQL database)
Send match results to player history database
Docker containers

Fig. 1. MatchOpt architecture

MatchOpt is the first open-source implementation of EOMM. Keeping in mind the player loads that can be experienced by a gaming server, MatchOpt is built on a scalable, distributed architecture. Fig. 1 shows the architecture of MatchOpt. The elements of MatchOpt are as follows.

a) REST API server: A public REST API server is exposed to the user, which they can use to join the player queue and wait for a match. This API server can be used in the future for player authentication and more. The server accepts join requests from players and moves them into the Kafka message queue.

b) Kafka Producer: A Kafka broker [2] is used consisting of two topics, one for the player queue and one for the matchmaking results. It is shown as two queues in Fig. 1 for ease of understanding. Although we are implementing EOMM, we group players into pools of similar skill ranges to ensure an approximately fair matchmaking.

c) Flink Matchmaker: We implement our core matchmaking strategy in Flink [3]. Players are grouped into pools every few seconds and EOMM is applied on these pools. First, a complete graph is constructed with the player pool, with the weight of the edge between players i, j being

$$c(s_i, s_j) + c(s_j, s_i)$$

where $c(s_i, s_j)$ is the *churn likelihood* of player i after playing a match with j, and s_i, s_j are the states of players i, j before the match. Similar to Chen et al., we also compute the outcome probability distribution $Pr(o_{ij})$ of the outcome of a match between s_i and s_j, and then use a known *churn prediction model* $c'(s_i, o_{i,j})$ to compute

$$c(s_i, s_j) = \sum_{o_{i,j} \in \{win, lose, draw\}} Pr(o_{i,j} | s_i, s_j) c'(s_i, o_{i,j}) \quad (1)$$

This computation can now make use of well established churn prediction studies that cannot take into account the presence of multiple players, but work well when the state of the player in question is provided.

d) PostgreSQL database: The churn prediction model of EOMM uses player histories such as win and loss streaks in its computations. Like most game developers, we assume the actual player histories are stored in cold storage and have the necessary player attributes for player state such as win and loss streaks stored in a Postgres database - again in line with most gaming services. The attributes for each player in the pool are fetched and used during churn prediction.

Now, **Minimum Weight Perfect Matching** is applied on the graph to collect player matched. A *perfect matching* on a complete graph ensures that no two edges have common vertices. Minimum weight ensures that the total churn of all these players is as low as possible. Thus, any algorithm performing MWPM can be used. We use the Edmonds' blossom algorithm implemented by Kolmogorov [7].

A. Implementation Choices

a) Message queues: We use Apache Kafka [8] for message queues. Kafka is a distributed event streaming platform designed for high-throughput, fault-tolerant, real-time data pipelines. It does a good job of handling large volumes of streaming data across distributed systems such as MatchOpt. It uses a publish-subscribe model, where producers send messages to topics and consumers subscribe to these topics to receive the messages.

Kafka's design allows each component of MatchOpt such as player queues and EOMM matchmaker to scale independently. It also integrates smoothly with Apache Flink [9], which we use for player processing and implementing EOMM.

While other messaging systems like RabbitMQ [10] also offer reliable message delivery and routing, we chose Kafka over RabbitMQ for several reasons. Kafka provides significantly better throughput, is optimized for distributed systems which need to be equipped for taking huge load, and supports replayable logs - a key feature for debugging, reprocessing, and ensuring consistency across matchmaking flows. Kafka's native integration with Flink makes our pipeline implementation easier, making it the choice for MatchOpt.

b) Stream Processing & EOMM: We use Apache Flink to implement EOMM in MatchOpt. Flink provides low-latency, high-throughput stream processing with support for stateful computation and event-time semantics. It provides exactly-once processing guarantees through its checkpointing mechanism and supports fault tolerance via pluggable state backends such as an embedded RocksDB store. In MatchOpt, we assign a subtask in Flink to each player pool partition from Kafka, computing the edge weights of our complete graph using churn prediction models and performing MWPM to obtain player matches. Flink also performs operator chaining to streamline execution and reduce overhead. The evolving player pool can be stored as state using RocksDB, enabling scalable and consistent matchmaking even under failures or restarts.

Other data processing frameworks like Apache Spark [11] and Apache Beam [12] also offer capabilities for distributed computation and stream processing. Spark operates on a micro-batch model, which introduces latency and limits its suitability for real-time, fine-grained operations like player matchmaking. Spark's state management is also relatively coarse-grained and primarily held in memory, which can be limiting under high-load scenarios in production. Apache Beam is a unified programming model for batch and stream processing. However, it requires external runners like Flink or Spark for execution, and its abstraction can limit access to perform optimized computations for matchmaking.

B. Load balancing & Fault tolerance

We perform a form of load balancing in MatchOpt through partitioning the players into player pools based on the skill range. We assume that the game developers using MatchOpt, like most gaming applications, compute a single rating for player skill such as ELO [7]. We use this skill rating to group players into player pools, to ensure approximately fair matchmaking even when our focus is on optimising engagement. Grouping players into such partitions also allows for load balancing, as more partitions can be allocated to pools which have a high number of players coming in. Most skill distributions of players in a game follow a bell curve, so allocating more partitions to the mid-skilled players (perhaps using custom hash functions) would help.

We make MatchOpt fault tolerant using the checkpointing mechanisms in Flink. Flink allows the user to store the state of the system using a mechanism similar to Chandy-Lamport's algorithm. This state can be stored in memory by default, or can be stored on a RocksDB state backend for larger loads. We employ retry mechanisms in with enough time between restarts to allow the system to recompute player matches for unmatched players.

The player pool is stored as the state of a subtask in MatchOpt. We chose to store the pool itself instead of the computed complete graph with edge weights because the graph computation and MWPM step are tightly coupled. While storing the graph would save its re-computation if the system has a failure, separating the graph creation step and MWPM step can introduce unnecessary latency, which is detrimental in everyday use with not many node failures. These two operators are already *chained* by Flink to improve efficiency. Storing just the complete graph as state also risks player pools which have not undergone the graph computation step to be lost. So we store just the raw player pool as state instead of having to store both.

IV. Results

When evaluating the effectiveness of MatchOpt as an Engagement Optimized Matchmaker, we tested its effectiveness across a range of player pool sizes, from 1,000 to 10,000 players. We wanted to see that MatchOpt would provide performance benefits similar to traditional skill-based matchmaking systems, while also seeing a decrease in player churn likelihood. For each test, we sent some number of pre-generated players that have a normal skill distribution to an API server that would send those players to our matchmaker and would then listen for the matchmaking results. We would then use those results to measure the system's matchmaking capabilities, the predicted player retention, the MMR (Matchmaking Rating) discrepancy, the matchmaking latency, and overall system throughput.

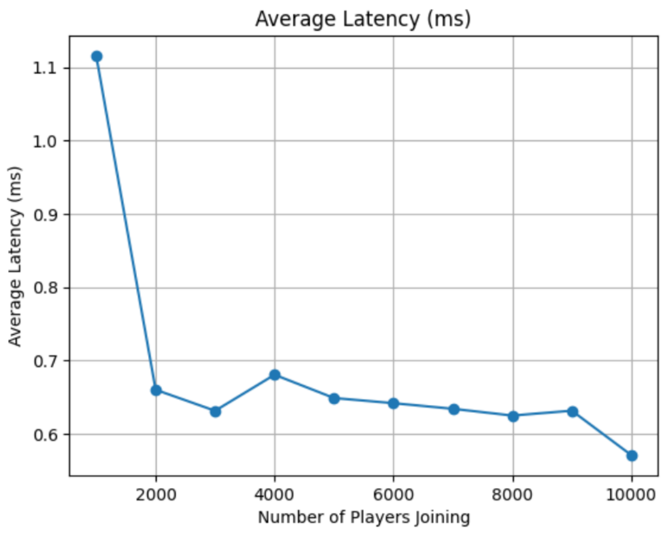

Fig. 2. Average Latency

For our matchmaking, we expected the number of matches to be equal to half the number of players joining. As expected, the total number of matches increased nearly linearly with the number of players, reaching 4,919 matches for 10,000 players. The players who were not matched represent the players whose matches with others did meet an optimal churn likeliness threshold. Match throughput per second remained stable, averaging around 58–61 matches/second across most configurations, which shows that our system does not degrade at scale. Interestingly, the average matchmaking latency per player decreased as the number of players increased. We saw a drop from 1.12 seconds at 1000 players to 0.57 seconds at 10,000 players. This counterintuitive result is likely due to the batching nature of the matchmaking process, since larger player pools allow for more efficient matching since there is a greater pool of players to match with, meaning faster matching potential and reducing the wait times. In terms of average MMR discrepancy, our system proved to be relatively consistent, hovering between an MMR discrepancy of 44 and 50. The standard deviation of MMR discrepancy generally decreased with the increasing player pool size as well, suggesting more consistent matchmaking when there is an increase in the number of players. Finally, the average churn likelihood fluctuated in a narrow band between 0.81 and 0.88, with no clear upward or downward trend. However, the standard deviation was highest at the smallest player pool, indicating that there is more variability in player experiences at lower volumes.

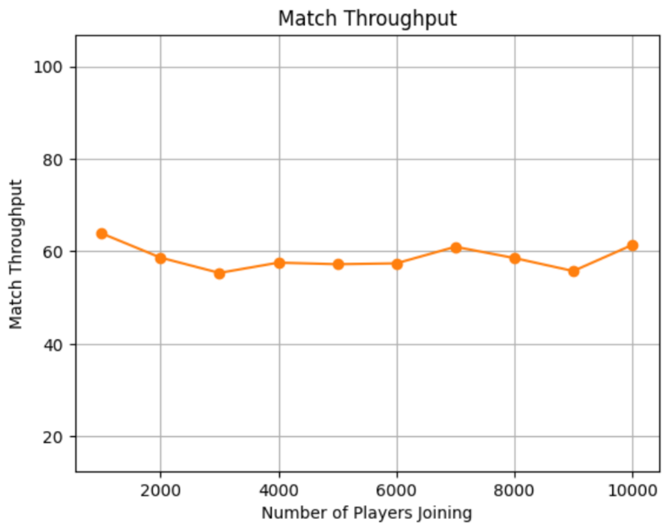

Fig. 3. Match Throughput

To evaluate the effectiveness of MatchOpt, we compared it with a traditional skill-based matchmaking approach by replacing our EOMM logic with SBMM-based matchmaking logic, which matched players within a close skill proximity and testing with the same 1000-player workload. While SBMM produced slightly more matches (496 vs 491) and a lower average MMR discrepancy (34.67 vs 49.36), the system showed an average churn likelihood of 0.9271, which when compared to MatchOpt's 0.8176, is an increase of 13%. This result corroborates the fact that EOMM-based systems can actively reduce the risk of churn by incorporating player engagement models rather than relying on skill similarity. Furthermore, EOMM showed only a marginal increase in latency per player (1.12 vs 1.09), suggesting that the more complex optimization did not substantially slow down matchmaking. However, SBMM did have a higher match throughput (89.61 vs 63.89 matches per second), likely due to its simpler matching criteria and faster execution speed. Overall, these results validate MatchOpts core objective: optimizing for long-term player engagement rather than immediate fairness. These results indicate that while EOMM-based matching does result in some skill precision and matching throughput degradation, it significantly reduces churn potential, thus making it better suited for sustaining healthy player populations over time.

TABLE I
COMPARISON OF EOMM AND SBMM AT 1000 PLAYERS

Metric	**EOMM**	**SBMM**
Total Matches	491	496
Average MMR Discrepancy	49.36	34.67
Std Dev MMR Discrepancy	66.60	24.76
Average Churn Likelihood	0.818	0.927
Std Dev Churn Likelihood	1.037	1.102
Average Latency per Player (s)	1.116	1.088
Match Throughput (matches/sec)	63.89	89.61

V. Limitations & Future Work

The current implementation of MatchOpt was deployed locally on isolated Docker containers for testing. The state is also maintained in-memory, which we have observed is sufficient for atleast 10,000 players joining the game. We plan to test MatchOpt on higher loads in a production environment by deploying the system on EC2 nodes. Once there, we can use Kubernetes for scaling the TaskManagers in Flink to run more subtasks in parallel. Going further, Cluster Autoscalers can be configured to add more EC2 nodes to the system under higher load. We plan to test our latency and throughput performance with OpenMatch after EC2 deployment.

Our vision is to develop MatchOpt into a complete matchmaking system with multiple options for the users to select the churn prediction model and matching strategy. We plan to have both pre-implemented strategies that can directly be used along with interfaces to allow users to provide their own strategies. MatchOpt can also be developed to be able to work across games, with multiple Flink jobs running for different games within the same matchmaking engine. We plan to explore these paths in the future.

VI. Conclusion

The landscape of matchmaking in games is rapidly evolving. While traditional skill-based systems remain dominant, new paradigms like EOMM promise to deliver significantly better player experiences and retention. However, there is a striking lack of scalable, extensible, and open-source systems that support these advanced approaches. We identify EOMM as the natural foundation for next-generation matchmaking systems due to its focus on long-term engagement. Yet current industry offerings—both commercial and open-source—fail to support it. There is a clear opportunity to develop infrastructure that bridges this gap: a modular, scalable, and transparent matchmaking system built around engagement optimization, not just fairness or latency.

Acknowledgment

This report was written for the CS 237: Distributed Systems Middleware course offered at UC Irvine from April-June 2025. We thank Prof. Nalini Venkatasubramanian for her approach and guidance throughout this class and for our project.

References

[1] A. E. Elo, *The Rating of Chessplayers, Past and Present*, New York, NY, USA: Arco Publishing, 1978.

[2] Z. Chen, S. Xue, J. Kolen, N. Aghdaie, K. A. Zaman, Y. Sun, and M. Seif El-Nasr, "EOMM: An engagement optimized matchmaking framework," *Proceedings of the 26th International Conference on World Wide Web (WWW)*, pp. 1533–1541, 2017, doi: 10.1145/3038912.3052559.

[3] Z. Deng, Y. Shi, Y. Jin, M. Wang, and Z. Zhang, "Globally optimized matchmaking in online games," *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2021, doi: 10.1145/3447548.3467074.

[4] Y. Sun, Z. Liu, J. Hu, J. Zhang, and C. Shen, "EnMatch: Matchmaking for better player engagement via neural combinatorial optimization," in Proceedings of the AAAI Conference on Artificial Intelligence, 2024

[5] G. Fan, C. Zhang, K. Wang, Y. Li, J. Chen, and Z. Xu, ”CUPID: Improving battle fairness and position satisfaction in online MOBA games with a re-matchmaking system,” *arXiv preprint arXiv:2406.19720*, 2024.

[6] Google Cloud, “Open Match: Flexible and extensible matchmaking for games,” in *Cloud Google Blog*, 2018.

[7] V. Kolmogorov, ”Blossom V: A new implementation of a minimum cost perfect matching algorithm,” *Mathematical Programming Computation*, vol. 1, pp. 43–67, 2009, doi: 10.1007/s12532-009-0002-8.

[8] J. Kreps, N. Narkhede, and J. Rao, ”Kafka: A distributed messaging system for log processing,” in *Proc. NetDB Workshop*, Athens, Greece, Jun. 2011.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, et al., ”Apache Flink: Stream and batch processing in a single engine,” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[10] ”RabbitMQ,” [Online]. Available: https://www.rabbitmq.com/. [Accessed: Jun. 13, 2025].

[11] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, and M. J. Franklin, ”Apache Spark: A unified engine for big data processing,” *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[12] ”Apache Beam,” [Online]. Available: https://beam.apache.org/. [Accessed: Jun. 13, 2025].