arXiv:2602.03189v1 [cs.DB] 3 Feb 2026

StreamShield：字节跳动 Apache Flink 经过生产验证的韧性解决方案

Yong Fang、Yuxing Han†、Meng Wang、Yifan Zhang、Yue Ma、Chi Zhang数据平台，字节跳动有限公司

摘要——**分布式流处理系统（DSPSs）是字节跳动实时处理和分析的骨干，其中 Apache Flink 驱动着全球规模最大的生产集群之一。确保韧性，即能够承受故障并快速恢复，同时保持运行稳定性，即在正常条件下提供一致且可预测的性能，对于满足严格的服务级别目标（SLO）至关重要。然而，由于集群规模、业务多样性以及显著的运维开销，在大规模生产环境中实现韧性和稳定性仍然充满挑战。本文介绍了 StreamShield，这是一套部署于字节跳动 Flink 集群中的、经过生产验证的韧性解决方案。StreamShield 从引擎和集群两个互补视角出发，引入了多项增强韧性的关键技术，涵盖运行时优化、细粒度容错、混合复制策略，以及外部系统下的高可用性。此外，StreamShield 提出了一套稳健的测试与部署流水线，确保生产版本发布的可靠性和稳健性。在生产集群上的大量评估表明，StreamShield 所提出的技术具备高效性和有效性。**

索引词—**分布式流处理、韧性、运行时优化、复制策略、高可用性**

I. 引言

在字节跳动，分布式流处理系统（DSPS）构成了实时数据处理和分析的支柱，为各种业务运营和面向用户的应用提供支持。通过接收并处理从用户、设备和服务收集的大量数据流，分布式流处理系统能够实现低延迟计算，并及时提取可操作的洞察。我们的生产环境采用 Apache Flink [1] 作为核心流处理引擎，运行着全球规模最大的分布式流处理集群之一。

在大规模、业务关键型流处理环境中，故障不可避免。在此类场景下，韧性 [2]–[4], 即抵御并快速恢复节点故障、网络分区和工作负载激增等不利事件的能力，对于达成严格的服务级别目标（SLO） [5]至关重要。确保生产环境中的 Flink 集群具备韧性，对于在不可预测的运行时条件下维持可用性、控制延迟并保障正确性至关重要。实现这一目标的另一项重要因素是运行稳定性，它能在正常运行条件下保持性能稳定且可预测，从而减少性能波动，并降低恢复干预的需求。

† Yuxing Han 博士是通讯作者，邮箱为 hanyuxing@bytedance.com。

然而，要保障字节跳动 Flink 集群的韧性，面临着诸多挑战，这些挑战源自其运行环境的规模、多样性和复杂性：**集群规模：在超大规模下管理海量有状态流处理**。我们的生产级流处理系统面临前所未有的可扩展性挑战。目前，Flink 集群支持超过 70,000 个并发流处理作业，并管理着 1,100 多万个资源槽位。在如此规模下，硬件故障、网络不稳定、软件缺陷和资源争用已成为常态，而非例外。这些中断通常表现为任务失败、任务延迟时间延长以及作业反复重启，进而共同降低全系统的处理稳定性。如何在这些频繁故障的情况下保障 Flink 作业的韧性，同时最大限度地减少运维开销并降低业务影响，是一项重大挑战。**业务多样性：具有不同 SLO 的异构工作负载：**Flink 集群支撑着各种内部业务应用，其工作负载各不相同。表 I 对业务场景进行了分类，展示了它们在恢复延迟和数据完整性方面的不同要求。关于恢复延迟和数据完整性。在所有业务场景中，亚秒级恢复且严格保证数据完整性是理想目标；但部分延迟关键型服务愿意放宽完整性保障，以换取极低延迟。营收关键型工作负载和数据同步管道要求完整交付数据，并在一分钟以内恢复，以保障业务正确性。相比之下，日志驱动的分析管道在相同延迟水平下可以容忍少量数据丢失。处于另一端的场景中，对于定时运营仪表板等离线数据仓库作业，只要严格保证完整性，小时级恢复便已足够；但在可容忍数据丢失的场景中，由于存在系统性故障的风险，这种方式便不可行。这些异构工作负载要求采用自适应韧性机制，在多样化的大规模流处理环境中平衡延迟与正确性。

运维开销：管理复杂的外部依赖。 在字节跳动大规模部署 Flink 的场景中，由于广泛依赖外部服务和基础设施组件，韧性管理变得尤为复杂。这些依赖涵盖 HDFS [6]、Apache ZooKeeper [7]、G¨odel [8], 等基础组件，并扩展至众多外部系统，共同构成了多层面的运行时环境。任何

数据完整性 / 恢复延迟	要求严格的数据完整性	可容忍少量数据丢失
亚秒级恢复	理想情况下适用于所有业务场景	延迟关键型服务 （例如，定向广告和实时推荐）
分钟内恢复	收入关键型服务（例如，变现指标：TikTok 打赏） 数据同步管道（例如，流式 ETL 工作负载）	日志驱动的分析管道 （例如，短视频互动统计：观看次数、点击次数）
小时级恢复	数据仓库作业（例如，定时运营仪表板）	不适用；容易发生系统故障

表 I：业务场景的异构性。

这些外部系统发生故障或出现性能异常，都可能波及 Flink 运行时，导致频繁的作业重启、任务故障转移和值班干预。实证数据表明，集群每天会经历数千次此类恢复事件，大幅增加了运维复杂性。要在这种情况下确保性能稳定，就需要先进的故障隔离和恢复机制，以有效减轻外部依赖带来的影响。

我们的方法。 为应对上述挑战，我们提出了 StreamShield，这是一种在字节跳动 Apache Flink 生产环境中经过验证的韧性解决方案，从引擎、集群和发布三个互补的 视角进行设计。从引擎视角（第三节）来看，StreamShield 通过自适应优化和细粒度容错 机制增强运行时韧性 ，从而缩小恢复范围。此外，它通过优化作业解析和任务部署来加速作业启动 ，并引入 HotUpdate 机制复用现有资源，以减少重启开销并提高部署效率。

在这些技术中，有些旨在解决异构工作负载下超大规模集群的通用韧性问题。例如，自适应 *Shuffle*支持动态且感知负载的数据重分布；自动扩缩容则会根据工作负载激增调整资源分配。其他技术则针对特定业务场景。例如，在允许少量数据丢失的场景中，*WeakHash*放宽了严格的键与任务绑定关系，将每个键映射到一组数量有限的候选任务，并动态选择执行任务，从而将热点键分散到多个任务中，缓解数据倾斜。与此同时，单任务恢复 将恢复范围从执行区域缩小到单个任务，使失败任务能够在不重启其相邻任务的情况下恢复，从而大幅降低恢复延迟，并尽可能减少对未受影响算子的干扰。

从集群角度（第 IV 节）来看，StreamShield 采用混合复制策略，以平衡恢复延迟与运维开销。它默认采用被动复制机制，并针对可用性要求严格的延迟关键型工作负载，有选择地切换为主动复制。除复制之外，StreamShield 还通过高可用配置、退避与重试策略以及多层容错技术，增强对外部依赖故障的稳健性。这些机制能够隔离来自外部环境的故障，防止级联中断，并在不同的基础设施动态变化下维持集群级稳定性。

从发布角度（第 V 节）来看，StreamShield 构建了一套稳健的测试与部署流水线。该流水线结合系统化的混沌测试 [9], [10] 和性能基准测试，验证新版本在易发生故障和高负载条件下的稳定性，确保其在生产环境发布前具备可靠性。通过衔接受控验证与实际部署，这套流水线不仅能验证系统在易发生故障和高负载条件下的韧性，还能通过自动化缺陷检测、防止代价高昂的部署后故障，降低运维开销。

总而言之，本文通过 StreamShield 的设计与部署做出了以下贡献：

- **引擎级韧性**：我们提出了相关技术，以加快作业启动速度，通过运行时优化动态调整执行，并借助细粒度恢复机制增强容错能力。
- **集群级韧性**：我们开发了一种结合被动复制与主动复制的混合复制策略，并为外部依赖提供高可用机制。
- **发布级韧性**：我们构建了一套全面的测试与部署流水线，集成混沌测试、微基准测试与宏基准测试以及在线探测任务。
- **实证验证**：在字节跳动开展大规模生产环境评估，表明 StreamShield 显著提升系统弹性、降低运维开销，并在异构且易发生故障的环境中保持稳定性能。

II. 预备知识

本节概述 Apache Flink 的架构与基本概念，随后给出分布式流处理中韧性问题的形式化定义。

A. Apache Flink

架构。 Apache Flink 是一个开源的分布式流处理框架，用于对无界和有界数据流执行高吞吐、低延迟且有状态的计算。如图 1 所示，其架构由三个核心组件组成：客户端、*JobManager*以及一组*TaskManager*。客户端将作业编译为逻辑数据流图 [11]–[13], ，并将作业提交给 JobManager，由 JobManager 将其转换为物理执行计划。随后，JobManager 通过查询已注册的 TaskManager 来识别可用资源，并将任务分配给合适的任务槽。部署指令下发后，TaskManager 会实例化相应的任务并开始处理

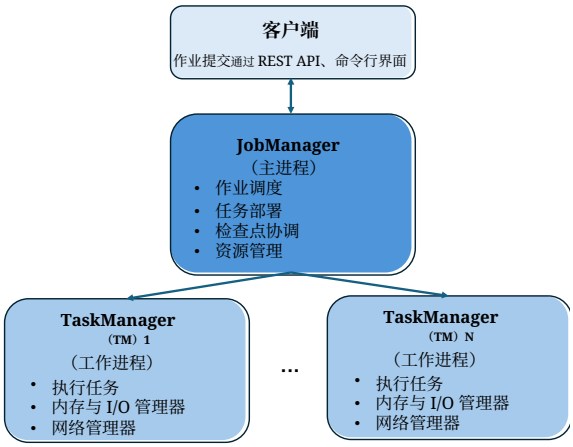


图 1：Apache Flink 的架构。

数据流，同时 JobManager 协调检查点机制并监控执行进度。

作业、算子和任务在 Flink 中，作业表示用户提交的完整流处理应用程序，以算子构成的逻辑数据流表示。算子表示定义逻辑处理语义的高级转换或计算（例如映射、过滤器、连接）。执行期间，每个算子会并行化为多个任务，这些任务是在 TaskManager 上调度的实际运行时单元。任务被分配到任务槽，接收输入流、维护本地状态并发出输出记录。因此，一个作业由算子构成的有向无环图组成，每个算子又会具体化为一个或多个任务，这些任务共同构成物理执行图。

检查点与保存点。Apache Flink 通过两个关键概念支持容错和状态管理：检查点和保存点。检查点会自动触发，属于轻量级且通常增量式的作业状态快照，用于在发生故障时进行恢复，确保恰好一次处理保证 [14]。相比之下，保存点由手动触发，是完整状态快照，会以独立于版本的方式存储在外部，主要用于作业升级、迁移或长期状态保留等运维场景 [15]。保存点通过提供更高的稳定性和灵活性来补充检查点，对于管理有状态 Flink 作业的生命周期至关重要。

B. 问题形式化

将分布式流处理作业 J 表示为有向无环图 $G = (V, E)$，其中每个顶点 $v \in V$都是一个算子实例（任务），每条边 $e \in E$ 表示算子之间的数据依赖。作业在一组物理节点 N上运行，每个节点承载一个或多个任务。我们将**韧性** 定义为系统在异常事件$E = \{e_1, e_2, ..., e_k\}$发生时维持服务等级目标合规性的能力，其中 e_i 表示不利的运行时状况，例如硬件故障、网络分区、软件崩溃或工作负载激增。

给定服务级别目标 $\mathcal{S} = (\gamma, \lambda_{max}, \tau_{max})$，其中$\gamma \in$ {full，partial} 表示工作负载是否要求完整的数据处理，还是可以容忍有限的数据丢失，λ_{max} 是可容忍的最大端到端延迟，τ_{max} 是异常事件发生后的最大允许恢复时间。我们的问题可以表述为：

定义 （韧性问题）。对于 J 在 N上运行期间发生的任意 $e_i \in E$，设计机制以确保：

$$completeness_requirement(J) = \gamma,$$
$$latency(J) \leq \lambda_{max},$$
$$recovery_time(J, e_i) \leq \tau_{max}$$

同时最大限度地降低额外资源开销和运营成本。

III. 韧性运行时与自适应执行

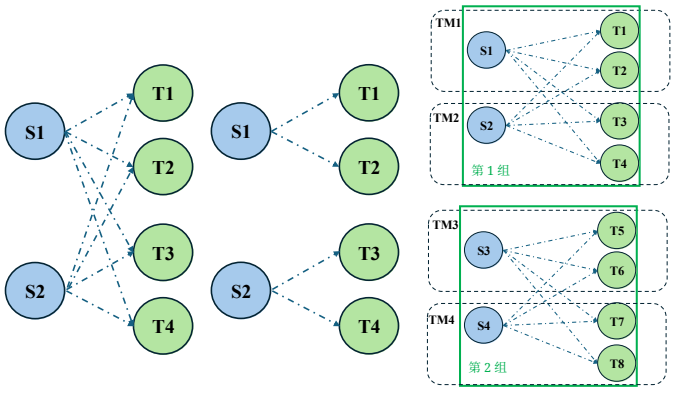
从引擎的角度来看，要在大规模流处理中实现韧性，不仅需要具备自适应运行时能力，以在动态工作负载下维持稳定执行，还需要健壮的容错机制。为此，StreamShield 引入了一系列运行时优化和细粒度容错机制。此外，我们还提出了多种加快作业启动的技术。

A. 运行时优化

为确保在三项关键约束下满足服务级目标，StreamShield采用了多种运行时优化技术，以适应动态工作负载、数据倾斜和波动的系统状况。具体而言，*AdaptiveShuffle*可动态平衡*WeakHash* 可缓解基于键分区时的数据倾斜，而*Autoscaling* 则会根据工作负载变化调整算子并行度。

基于积压的 Shuffle。Flink提供了两种在并行任务间重新分配数据的自适应 Shuffle 策略：*Re-balance*和*Rescale*。*Rebalance* 策略（图 2a）以轮询方式将记录均匀地重新分配到所有下游子任务中，无论算子并行度配置如何，都能实现均匀的负载分布。相比之下，*Rescale*策略（图 2b）会将数据交换限制在与上游并行结构相匹配的局部下游子任务子集中；每个上游子任务只与固定范围内的下游子任务通信。

由于 Rebalance 和 Rescale 都使用轮询算法分发记录，因此它们默认下游处理能力是同质的。然而，在实际部署中，由于节点异构和工作负载不均衡，这一假设往往不成立 [16]。为解决这一局限，我们提出基于积压的 ***Shuffle***，这是一种动态且感知下游负载的数据重分布策略，基于 Flink 的基于信用的流量控制机制 [17],：每个下游任务向其对应的上游任务发放数量有限的缓冲区信用，以调节传输。在基于积压的Shuffle中，每个通道（即上游任务与下游任务之间的物理通信路径）都设有可配置的信用阈值；一旦



(a) Rebalance。 (b) Rescale。 (c) Group-Rescale。

图 2：洗牌策略优化。

阈值表明出现过载，数据便会从拥塞通道转移出去。通过主动将记录引导至利用率较低的节点，这种自适应路由策略能够缓解反压传播，增强检查点可靠性，并提升整体系统稳定性和资源利用率。**Group-Rescale**。 为提高 Rescale 策略下上游到下游数据重分布的灵活性，我们提出了一种名为*Group-Rescale*的优化方案。与标准 Rescale 不同，后者仅根据并行度对齐情况，将每个上游任务的数据发送至固定的下游任务子集；Group-Rescale 采用基于分组的通信范式。具体而言，上游任务和下游任务会被划分为多个互不相交的组，每组包含部分上游任务和下游任务。在每个组内，所有上游任务都与所有下游任务完全互连，且数据洗牌严格限制在组内。

该策略有效解决了同一 TaskManager 上协同部署的任务共同成为慢任务的情况。如图 2c 所示，当任务 S1、T1 和 T2 部署在同一个较慢的 TaskManager 上时，基于积压的 Shuffle 机制会失效，因为所有协同部署的任务都会承受类似的反压。借助 Group-Rescale，数据可以在同一组内的多个 TaskManager 之间进行重分布（例如，第 1 组中的 TM1 和 TM2 ），从而将通信范围扩展到单个节点之外。这使健康的上游任务能够绕过执行缓慢的下游任务，实现更均衡的负载分布，并防止局部瓶颈蔓延至整个数据流。

从概念上讲，Group-Rescale 在 Rebalance 与 Rescale 之间取得了平衡。Rebalance 提供了完全灵活的数据重分布能力，但代价是增加网络开销和调度开销；Rescale 则将通信开销降至最低，但适应能力有限。Group-Rescale 介于两者之间：它保留了 Rescale 的轻量级协调方式，同时适度放宽其通信限制，从而支持组内受控的跨 TaskManager 交换。

WeakHash. Flink 中传统的基于键的分区会将所有具有相同键的记录路由到同一个下游任务，从而确保强一致性。虽然这种设计对于需要确定性聚合的有状态流处理算子是必不可少的，但当少数热点键主导输入分布时，往往会导致严重的负载不均衡。这种数据倾斜会造成处理延迟、积压累积和资源利用不足，最终降低大规模流处理作业的端到端性能。

为缓解这一情况，StreamShield 引入了*WeakHash*，这是一种分区策略。鉴于某些场景可以容忍少量数据丢失，该策略有意放宽了严格的键与任务绑定。WeakHash 的一个显著特点是会为每条记录动态选择执行目标。WeakHash 不会将每个键确定性地映射到单个 TaskManager，而是将该键关联到一组有限的 候选 TaskManager，并通过轻量级调度策略（例如根据运行时指标进行负载感知选择）选出实际的下游任务。这种设计将数据倾斜严重的键所产生的负载分散到多个任务，从而避免热点并改善整体工作负载均衡。

除了允许数据丢失的场景外，WeakHash 在涉及维度表查找的数据仓库工作负载中也尤其有效。在这些工作负载中，事实流会通过分布式缓存或外部键值存储（例如 ABase [18]）获取辅助属性。此类工作负载极易受到键倾斜的影响，因为少数热门维度键（例如频繁访问的用户或产品 ID）可能主导查询分布。请注意，维度查找通常具有幂等性，且不会修改外部表。因此，放宽针对每个键的一致性要求不会损害正确性。相反，这种做法能将存储访问压力分散到多个连接，并在严重数据倾斜时稳定查询延迟。

自动扩缩容。大规模流处理工作负载呈现出高度动态的流量模式，通常受用户活动波动、昼夜周期或突发事件驱动。根据峰值需求预先配置过多资源会造成浪费并导致利用率偏低，而资源配置不足则会因积压增加和延迟升高而降低服务质量。传统自动扩缩容方法 [5], [19]–[21]依赖 CPU 或吞吐量等粗粒度指标，无法捕捉输入速率、算子选择率和状态大小之间复杂的相互作用 [22]–[24]。这种不匹配往往导致响应式扩缩容决策要么落后于工作负载变化，要么在资源过度配置与配置不足之间反复摇摆。StreamShield 采用了受 DS2模型 [22],[25]启发的自动扩缩容策略。DS2 并非仅根据资源利用率进行扩缩容，而是通过分析输入速率、处理速率、积压增长和算子扇出等算子级信号，识别扩缩容瓶颈。

然而，要在生产级环境中应用 DS2，必须解决其原始设计的两个关键局限：依赖稳定的指标，以及假设可以不受约束地重新配置。首先，为了提高模型在瞬时波动或工作负载倾斜导致指标失真时的准确性，StreamShield 引入了平滑与补偿策略。具体而言，对短期指标振荡进行平均处理，并通过校正因子调整数据源的繁忙时间估算，同时在反压导致繁忙时间信号饱和时，以实际处理速率取而代之。

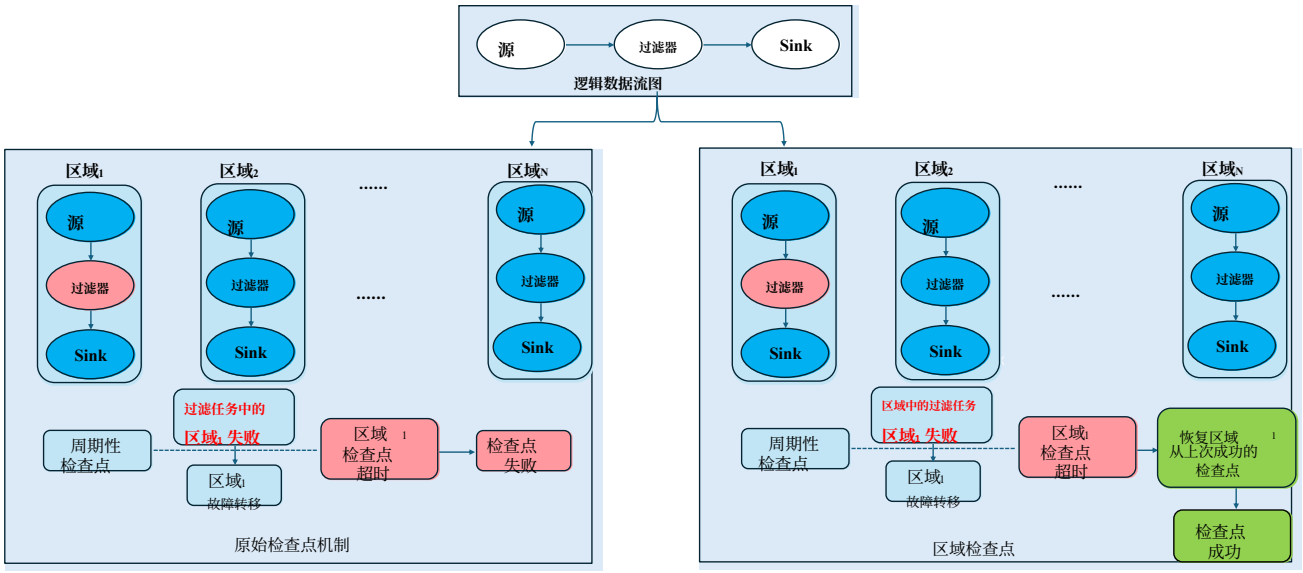


图3：原始方式与区域检查点机制。

其次，为了进一步提高稳定性，StreamShield 集成了回滚和安全机制。这些机制包括在调整失败时自动回滚、业务驱动的缩容策略（例如，在高峰时段禁用缩容），以及速率限制和故障转移感知型熔断等保护措施，以避免级联效应。总体而言，这些改进克服了 DS2 对噪声指标的敏感性，以及在大规模重新配置时的不稳定性，从而使其能够在生产级流式部署中可靠运行。

虽然 DS2 在大多数场景中表现出色，但面对更复杂的流处理作业流拓扑时，它往往无法实现最佳的算子并行度。作为一种自然的扩展，我们设想集成 StreamTune [26],，这是一种基于学习的自动扩缩框架，可通过历史工作负载轨迹和在线反馈持续优化扩缩策略。与启发式或基于规则的方法不同，StreamTune 采用预训练与微调框架，利用历史执行数据中的全局知识，为特定作业进行并行度调优。值得注意的是，它利用图神经网络 [27]–[29] 捕捉算子并行度、DAG结构与已识别的算子级瓶颈之间的复杂关联。

B. 细粒度容错

Apache Flink 中的传统容错策略虽能保证正确性，但由于粒度较粗，在大规模有状态工作负载中往往会导致较长的恢复延迟和不必要的中断。为增强生产流集群的韧性，我们在 StreamShield 中引入了一套细粒度容错机制，以改进恢复过程。

区域检查点。 在大规模生产环境中，Apache Flink 原始的全局检查点机制往往会成为制约系统弹性的因素。例如，在典型的数据同步工作负载中，单个流处理作业可能包含数万个任务，每个任务处理来自上游消息队列（如 ByteMQ）的一个独立分区 [30]。假设每五分钟触发一次检查点，且每个任务发生故障的概率为 0.0001，那么预计每个检查点间隔内至少会有一个任务发生故障。在 Flink 原始的检查点方案下，任何单个任务发生故障都会导致正在进行的检查点被中止。这一行为会导致大型流处理作业频繁出现检查点不稳定性。由此造成的低效不仅浪费计算资源，还会削弱系统弹性，增加恢复延迟，并提高违反 SLO 的风险。

为克服这一局限，StreamShield 引入了区域检查点机制，通过按执行区域划分检查点来隔离局部故障[1]。该机制非常适合规模大、异构性强，且具有高算子并行度和不同状态大小的流处理作业。具体而言，当某个区域的检查点失败时，系统会自动将该区域最新的成功检查点与其他区域当前的成功检查点合并，从而生成一致的全局检查点。图3展示了任务失败时，原始检查点机制与区域检查点机制在检查点行为上的差异。从包含三个算子的逻辑数据流图开始，物理执行图被划分为 N 个互不相交的区域。当 Region_1 中的某个任务失败时，原始检查点方案（左）和区域检查点机制（右）都会触发任务故障转移和作业恢复。关键区别在于检查点协调机制如何处理部分故障。在原始方案中，由于 Region_1 无法完成其任务级检查点，整个

[1]此处，区域表示作业执行图中的一个故障恢复单元，定义为由阻塞式数据交换所界定的一组互连任务，使得同一区域内的任务在恢复时相互依赖。

检查点尝试被标记为失败。相比之下，在区域检查点机制中，系统会自动将 Region_1 最新的成功检查点状态与其他区域（例如从 Region_2 到 Region_N）的检查点状态合并，从而得到全局成功检查点。该机制通过防止单个区域故障导致整个检查点失效，有效提高了整体检查点成功率。

单任务恢复。

为了缩小恢复范围，Apache Flink 社区引入了区域故障转移策略 [31], ，该策略仅重启失败区域及其依赖算子，而非整个作业。尽管区域故障转移策略能够保证全局状态一致性，但仍可能带来显著的恢复延迟。在可接受少量数据丢失的延迟敏感场景中，这种粗粒度方法还可能进一步引入明显的服务抖动，对原本未受影响的算子造成不必要的干扰。

为应对此类场景，StreamShield 引入了一种名为 单任务恢复 的机制。该机制通过将恢复范围限定在失败任务，而不向其所在区域级联重启，解决了上述低效问题。一旦检测到任务失败，JobManager 仅对该任务启动恢复，而拓扑中的所有其他任务继续处理。这一设计有效地将故障处理与全局故障转移链解耦，从而同时缩小重启范围并降低状态恢复工作量。需要注意的是，该机制以放弃严格的精确一次保证为代价，换取更短的停机时间和更低的运维影响，因此非常适合一致性要求较为宽松的延迟敏感应用。

从技术上讲，该机制重新定义了失败任务与其相邻任务之间的通信协议。对于上游算子，在任务停机期间，所有发往失败任务的记录都会被丢弃；任务恢复后，则会重新建立正常的数据传输。对于下游算子，我们会丢弃故障发生前产生的所有不完整或部分输出，确保后续处理从干净的状态恢复。这种选择性数据清除和通道重连使失败任务能够重新加入作业，而不会触发同一区域内所有任务的故障转移。此外，它还能最大限度地减少对正在进行的计算的干扰，同时保持可预测的算子级语义。

单任务恢复对于大规模推荐流水线中的样本拼接工作负载尤为有效：数据片段通过键哈希进行关联，以重构底层机器学习模型所需的完整训练样本。由于少量样本损失对推荐模型准确率的影响可忽略不计，单任务恢复在延迟与正确性之间实现了均衡的权衡。来自生产部署的实证证据表明，与区域故障转移相比，该机制可将恢复延迟降低数个数量级，缓解积压累积，并在频繁的局部故障下稳定吞吐量。因此，它为在低延迟流处理环境中持续满足 SLO 提供了一条实用且资源高效的路径。

表 II：Nexmark Q2 的作业启动开销

TM 数量	作业解析 (ms)	资源分配 (ms)	任务部署 (ms)
512	315	234,977	9,446
1024	268	338,615	15,565
2048	800	**500,889**	**35,476**

状态懒加载。 在 Flink 的传统恢复工作流中，算子状态需先从远程存储下载检查点数据，并重新初始化相应的状态后端（例如 RocksDB [32]），数据处理才能恢复。当状态大小较大，或远程存储系统出现慢任务时，这种急切状态恢复模型往往会导致显著的停机时间。

为了解决这一限制，StreamShield 引入了状态懒加载，将作业恢复执行与完整的状态物化解耦。算子无需等待所有状态都在本地就绪后才开始恢复，而是先利用部分状态恢复执行，其余部分则在后台异步恢复。从技术上讲，这一机制将恢复转变为按需驱动过程。任务重启后，算子会迅速重新建立数据流管道，并利用本地已可用的状态子集恢复处理。当访问尚未物化的状态条目时，系统会触发从远程存储异步获取数据，从而确保正确性，同时避免阻塞执行。通过将状态恢复与正常处理并行进行，该机制最大限度地缩短停机时间并减少积压增长。

状态懒加载尤其适用于状态规模庞大的作业，其状态大小可达数十太字节；它也适用于异构存储环境，在这类环境中，掉队节点会拖慢恢复速度。在这些场景下，该机制能够显著减少恢复导致的中断，从而提升金融交易、在线推荐流水线和实时监控系统等延迟敏感工作负载的韧性。

C. 作业启动加速

为满足严格的延迟与恢复时间要求，StreamShield 引入了一套机制，以加速作业启动与重启。在生产流处理环境中，作业重启通常由异常事件（如节点故障）或计划维护（如业务逻辑更新和系统升级）触发。更快的启动可缩短作业不可用或处于降级状态的时长，从而降低违反服务等级目标的风险。

表 II 报告了 Nexmark Q2 [33] 在每个 TM 并行度固定为 2 的情况下，随着 TM 数量变化，作业解析、资源分配和任务部署各阶段的作业启动开销。三个阶段都会带来明显延迟：作业解析耗时数百毫秒，而随着集群规模扩大，资源分配和任务部署产生的延迟也日益显著。在 2048 个 TaskManager 时，总启动延迟增至数百秒。这些结果表明，作业启动仍然开销高昂，也揭示了大规模环境中存在显著的优化空间。

优化作业解析和部署。为降低作业启动开销，StreamShield 在作业解析和任务部署阶段均采用了优化措施。在作业解析期间，JobManager 引入了内存对象复用机制，以避免重复实例化执行计划对象，重点处理数据流图中表示任务间数据通信的边对象。该机制会识别物理执行计划中相同或语义相似的边对象，例如采用相同分区方案或相同混洗策略的边对象，并复用这些对象，而不是创建新实例。除了减少执行计划对象的总数外，该方法还通过整合元数据和任务描述信息，缩减单个对象的内存占用，从而降低后续部署阶段的序列化开销。

在任务部署阶段，传统方法通常会分别将计算任务分派给各个TaskManager（TM），这会产生较大的通信开销和延迟。为缓解这一低效问题，StreamShield采用一种批处理策略：JobManager会将分配给同一TaskManager的多个任务的部署请求汇总为一条整合消息，然后再进行发送。通过将发往同一TaskManager的所有任务部署描述符进行批处理，该方法显著减少了JobManager与TaskManager之间的远程过程调用（RPC）次数和网络往返，从而降低通信开销，加快大规模作业部署。

缓解慢启动TaskManager。在作业部署期间，慢启动TaskManager可能会导致作业启动严重延迟。JobManager必须先为所有所需的TaskManager分配并部署任务，然后才能将作业转换到下一个运行中状态。如果作业未能及时获得足够的资源，且TaskManager分配被延迟，则整体作业启动延迟将会增加。

为缓解这一问题，StreamShield 采用两步策略。第一步，它监控注册延迟和槽位上报时间等指标，以识别启动缓慢的 TaskManager，并将耗时显著超过其他同类节点的 TaskManager 判定为慢任务。只有当 TaskManager 分配时间超过可配置的阈值（例如 2 分钟）时，才会触发第二步：JobManager 主动额外分配最多 30% 的 TaskManager，且数量不超过可配置的上限（例如 5 个），确保有足够的资源，而无需等待所有启动缓慢的 TaskManager 完成初始化。作业进入运行状态后，这些冗余 TM 会立即释放，以减少资源开销。这项优化在容器化 环境中特别有效，因为底层节点的性能瓶颈会拖慢任务初始化，例如物理机器上的 I/O饱和会延迟容器镜像下载，进而延长作业启动时间。

HotUpdate。为进一步加快作业重启，StreamShield 引入了 HotUpdate 机制，专为业务逻辑频繁更新的场景设计。在传统重启过程中，JobManager 会取消旧作业，释放 TaskManager 槽位，并在重新部署前重新获取资源，这可能导致显著的启动延迟。HotUpdate 通过复用已分配的资源，而非彻底拆除并重新初始化执行环境，消除了这部分开销。具体而言，旧作业会直接在现有 TaskManager 上停止，新作业则部署到相同的运行进程中，从而保留槽位分配和 JVM 预热状态。生产环境中的实证表明，结合其他启动加速技术后，HotUpdate 可将作业重启延迟缩短至 20 秒。

从韧性的角度来看，HotUpdate 机制增强了系统应对运行中断的能力，尤其是在频繁更新业务逻辑或进行系统维护时。通过消除完整重新部署所带来的停机时间，它缩短了潜在数据丢失或积压累积的风险窗口。此外，它还支持以最小的干扰对业务逻辑进行滚动升级，减少对上游和下游系统的影响，从而有效弥合大规模流处理环境中高可用性与快速迭代之间的差距。因此，HotUpdate 是在生产级 Flink 集群中维持服务稳定性的有力工具。

IV. 高可用性构建

要在大规模 Flink 集群中实现高可用性（HA），需要采用覆盖执行层及其外部依赖的容错机制。一方面，复制策略决定作业从任务和算子故障中恢复的速度，并在不同异构工作负载之间权衡恢复延迟与资源效率。另一方面，HDFS、ZooKeeper、G¨odel 等关键外部系统的可靠性会直接影响 Flink 保持精确一次语义和持续服务的能力。本节介绍 StreamShield 的高可用性构建方案，该方案将混合复制与依赖感知容错相结合，以确保生产环境中的运行具备韧性且不中断。

A. 复制策略

复制策略是实现数据流处理系统高可用性和高韧性的关键。典型策略有两种，分别是主动复制和被动复制。主动复制 [34], [35] 会为一个算子维护多个并发副本，每个副本并行处理相同的输入流。该策略可实现快速故障转移，恢复延迟几乎可以忽略不计，因为发生故障时，备用副本可立即接管执行。主动复制的主要缺点是资源开销巨大，而且要求算子具有确定性行为，以确保副本之间的一致性。

与主动复制不同，被动复制 [36]–[38] 通过维护备用副本实现高可用，资源开销更低；这些副本在发生故障前一直处于空闲状态。它们不会与主副本并行执行，而是定期接收状态更新，并在恢复过程中应用这些更新。

这种方法虽然节省计算资源，但恢复延迟更长，因为失败的算子必须先重新启动并重新加载其状态，才能恢复执行。尽管存在这一缺点，对于不要求严格低延迟恢复、但资源效率至关重要的工作负载而言，被动复制仍是一种实用策略。

与先前工作中的类似思路 [39]–[41],相呼应，StreamShield 采用混合方法，在不同业务场景中平衡恢复延迟与资源效率。基于 Flink 对周期性检查点的原生支持，被动复制作为我们的默认机制，为仪表盘监控和分析流水线等优先级较低的工作负载提供容错能力，同时将资源开销降至最低。对于对延迟敏感且有严格可用性要求的应用，StreamShield 会有选择地通过预先部署的次级副本切换到主动复制，以确保快速故障转移和服务连续性。通过根据工作负载优先级匹配复制模式，这种混合策略在效率与响应速度之间实现了实用的平衡，从而增强了大规模流处理系统的整体韧性。

B. 外部依赖的容错

作为一个有状态流处理框架，我们内部的 Flink 集群依赖多个外部系统来实现容错和高可用性。核心依赖包括用于持久化状态管理的 HDFS [42], [43], 、用于集群协调和 JobManager 领导者选举的 Apache ZooKeeper [7], [44], ，以及用于资源编排和任务部署的 G¨odel [8], [45]。这些组件与 Flink 的执行流水线紧密集成，因此其故障可能会直接中断作业调度，或损害集群可用性。与通常可通过局部恢复缓解的暂时性任务失败不同，这些依赖系统发生故障会造成系统性中断，威胁所有正在运行的作业的可靠性。在大规模生产环境中，诊断和处理此类故障的复杂性会进一步加剧，因为运行依赖跨越存储、协调和调度等多个层级。本小节分析了这些系统在我们的生产部署中的故障模式，并介绍了为减轻其影响而开发的容错机制。

HDFS 容错：HDFS 是 Flink 检查点和保存点的重要持久化存储，其可靠性是系统韧性的基础。典型故障模式包括 NameNode 崩溃和 DataNode 不可用。这些故障会直接导致检查点超时，引发对齐延迟并造成反压，最终导致作业反复重启。

为降低这些风险，StreamShield 要求以高可用配置部署 HDFS，采用主备NameNode，并在多个 DataNode 之间进行复制，以避免单点故障。为应对长尾存储延迟和局部故障，StreamShield 将增量检查点与区域级恢复和惰性状态恢复相结合，从而减少如第 III-B 节所述，在故障转移期间缩小恢复范围并减轻 I/O 压力。此外，为 HDFS 增加其他持久化存储后端（例如对象存储），可增强系统抵御 HDFS 长时间中断的能力。最后，系统还纳入持续健康监控和自动故障转移流程，以便及时检测异常并触发快速恢复操作，从而增强整体系统弹性和稳健性。

ZooKeeper 容错：Apache ZooKeeper 是 Flink 的核心协调服务，支持集群元数据管理和领导者选举。其稳定性会直接影响 JobManager 的高可用和任务调度。然而，它容易受到会话过期、仲裁组丢失、领导者选举风暴以及服务器过载时高延迟操作等故障的影响。这些问题可能导致 JobManager 领导权频繁变更、作业提交停滞或任务失败被误判，从而威胁集群韧性。

为降低对 ZooKeeper 的强依赖，StreamShield 引入了一种轻量级容错机制，防止运行中的作业在 ZooKeeper 不可用时失败。具体而言，除了在 ZooKeeper 中保存领导者元数据外，StreamShield 还会在 HDFS 中维护一份冗余副本。检测到 ZooKeeper 故障后，系统会自动回退到 HDFS 副本以获取领导者信息，确保作业执行不间断。只有在 ZooKeeper 和 HDFS 均不可用，或 HDFS 中存储的领导者信息与内存状态不一致时，才会终止受影响的作业以确保正确性。

G¨odel 容错：G¨odel 是 Flink 的资源编排和任务部署框架，负责分配计算资源、启动 TaskManager，并在运行时动态调整算子并行度。它的可靠性对于确保作业及时初始化和任务稳定执行至关重要。典型故障模式包括调度器不可用或限流，以及过时或不一致的资源视图导致资源错配。这些故障可能表现为任务部署延迟、恢复时间延长，或慢启动的 TM 导致启动延迟增加。

为降低这些风险，StreamShield 采用了多层容错机制。在引擎层面，StreamShield 集成了 HotUpdate 等优化，使作业无需释放现有资源即可重启；还采用了慢启动TM缓解技术，分配多达 30% 的冗余TM来吸收启动延迟，并在作业运行后立即释放（见第 III-C 节）。在控制平面层面，部署冗余编排端点和回退策略，以掩盖 G¨odel 的暂时性故障 [25]。部署请求采用指数退避 [46] 和幂等重试 [47]，以避免在不稳定情况下重复提交。具体而言，当作业提交因 G¨odel 暂时不可用而失败时，StreamShield 会自动采用指数退避进行重试，并执行作业唯一性校验，以防止重复提交导致重复执行。具体来说，指数退避通过拉开重试间隔，避免在暂时性故障期间造成过大负载

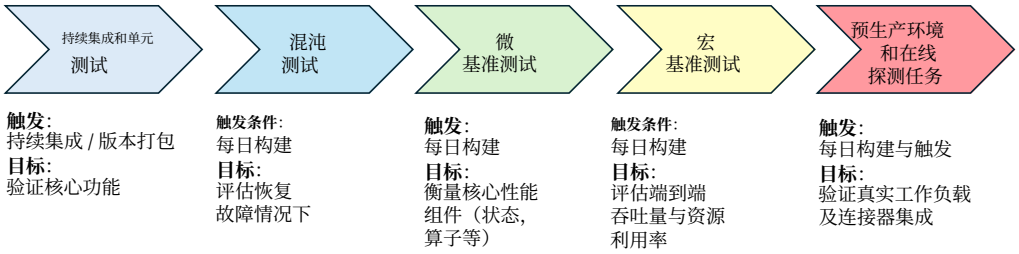


图4：Flink 引擎的测试与发布流水线。

；而幂等重试则确保重复请求不会产生重复效果，从而保证正确性。此外，系统还会通过健康检查和持续监控检测调度器状态是否降级，并触发自动修复。总体而言，这些机制可确保编排层的不稳定不会演变为系统性中断，从而保障大规模 Flink 部署的韧性。

V. 稳健测试与发布流水线

从工程角度来看，提升流处理引擎的韧性还需要对其关键组件进行系统性验证。为此，StreamShield 采用稳健的测试与发布流水线，持续让引擎经受易发生故障和高负载条件的考验，从而及早发现潜在缺陷并确保稳定发布。通过在生产部署前发现漏洞，该流水线增强了系统抵御运行时故障的能力。本节将介绍该流水线的概况，并进一步详细讨论两种代表性的测试方法：混沌测试和性能测试。

A. 概述

图4展示了 StreamShield 的 Flink 引擎测试与发布流水线，该流水线集成了多层验证，以确保功能正确性和生产可靠性。该流程从持续集成（*CI*）和单元测试开始，在版本打包后触发；自动化测试套件会验证核心功能并确保向后兼容性。在此基础上，混沌测试会在每日构建中执行，以检验系统在各种故障场景下的韧性。

为补充故障韧性验证，我们定期开展微基准测试，衡量算子执行和状态后端等核心组件的性能。这些细粒度基准测试有助于深入了解关键组件的效率，并发现代码变更可能引入的性能回归。与此同时，宏基准测试通过在集群规模下部署合成流查询进行宏观评估，从而能够在真实工作负载下衡量端到端吞吐量和资源利用率。

最后，预生产环境和在线探测任务会在每日构建中持续执行，以验证引擎在真实业务场景下的稳定性，以及与外部依赖的互操作性。探测任务是轻量级的流处理作业，可模拟具有代表性的工作负载并与外部服务交互，从而在不产生显著开销的情况下，检验端到端数据路径。此阶段衔接了受控实验测试与实际生产环境，确保每次发布在大规模部署前，既满足系统级性能目标，也符合业务级韧性要求。

B. 混沌测试

混沌测试是分布式系统中的一种系统化故障注入方法，通过有意引入可控故障，评估系统应对意外中断的韧性，以及在不利条件下维持服务连续性的能力。在 StreamShield 中，混沌测试是验证 Flink 容错机制的关键手段。在硬件层，我们模拟网络延迟增加、带宽限制、CPU干扰和磁盘 I/O 争用等异常，以捕捉资源异构性和暂时性基础设施瓶颈所造成的影响。在进程层，我们会在作业执行期间主动终止 TaskManager 和 JobManager 实例，以模拟节点崩溃或容器驱逐等真实故障场景。这些测试系统地检验系统从检查点恢复执行状态、重新分配资源并恢复算子流水线，同时不违背正确性保证的能力。通过结合硬件层面的扰动与进程层面的故障，混沌测试全面覆盖暂时性和灾难性中断，并深入揭示大规模分布式流处理系统部署中的恢复延迟、检查点可靠性，以及端到端作业语义的保持情况。

C. 性能测试

为确保系统稳健性，并防止 Flink 升级期间出现性能回归，StreamShield 建立了一套结构化的性能测试框架，涵盖组件级和系统级评估。该框架分为两个互补类别：微基准测试和宏基准测试，分别用于不同的诊断与验证目的。

微基准测试：微基准测试聚焦于 Flink 引擎的关键计算路径。具体而言，我们评估三个主要方面：算子执行、作业调度和状态管理。特别是对于状态管理，我们研究读写操作的延迟、大规模键空间扩展下的可扩展性（即算子状态中维护的不同键数量大幅增长的场景

表 III：工作负载选择。

Job	描述	逻辑数据流图	源速率
Nexmark Q2	过滤满足预定义条件的出价记录。	两个逻辑节点 包含一个源	固定为 0.8 M/s
Nexmark Q12	在固定的处理时间窗口内统计每位用户的出价次数	三个逻辑节点 使用一个源	固定为 0.8 M/s
数据同步（DS）	从消息队列向 Apache Hive 写入记录	两个逻辑节点 使用一个源	介于 1 M/s-6.4 M/s
样本拼接（SS）	基于在线推荐结果对样本进行实时拼接 推荐结果	数十个逻辑节点 带有两个数据源	变化范围介于 5 千条/秒-45 千条/秒

），以及在各种检查点配置下的恢复效率。所有评估均在单节点环境中进行，以最大限度地减少外部变异并确保可复现性。主要目标是在性能回归影响端到端工作负载之前，识别各个组件中的性能回归。

宏基准测试：宏基准测试对 Flink 在近似生产级流处理应用的工作负载场景下的性能进行端到端评估。在 StreamShield 中，我们采用 Nexmark 套件 [33], ，其中包含一组以拍卖系统为模型的连续查询。这些查询涵盖多种处理模式，包括源数据摄取、过滤、聚合，以及更复杂的基于窗口和连接的操作。基准测试在多节点 Flink 集群上执行，并持续监控 CPU 利用率和查询吞吐量等系统级指标。收集到的结果通过交互式仪表板进行分析，以捕捉工作负载动态，并指导性能调优与发布验证。

VI. 实验评估

本节介绍在字节跳动生产环境中对 StreamShield 进行的性能评估。我们聚焦于具有代表性的面向引擎的机制，以展示 StreamShield 的效率与有效性。我们首先介绍实验设置，然后考察每项所提技术的影响。

A. 实验设置

环境。我们在字节跳动部署的由 G¨odel 管理的 Flink v1.11 集群上部署 StreamShield。该集群配备 384 个 AMD Genoa CPU 核心、2.3 TB DRAM 和 15 TB NVMe SSD 存储，并通过双路 100 Gbps 网络接口互连。Flink 为 JobManager 和 TaskManager 均配置了 16 GB 进程内存，其中每个 TaskManager 分配四个 CPU 核心和四个任务槽。系统采用 RocksDB 状态后端和增量检查点机制，每 60 秒执行一次检查点。

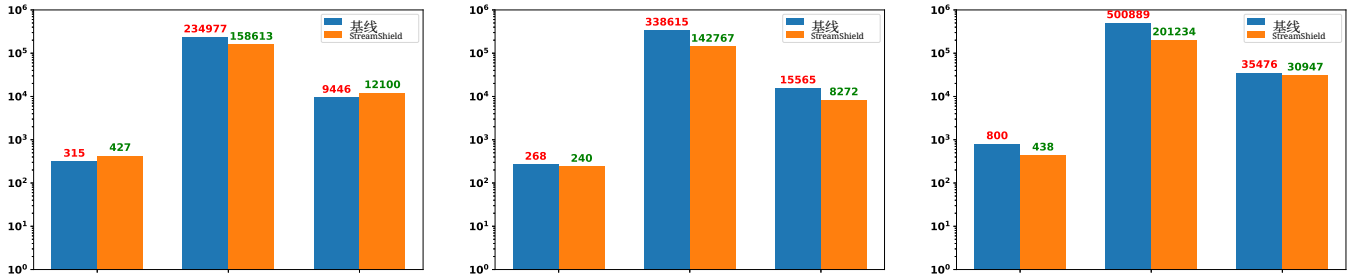
工作负载。表 III 总结了评估中使用的代表性工作负载。此处，K/s 和 M/s 分别表示每秒数千条和数百万条记录。Nexmark Q2 包含一个简单的无状态过滤器，而 Q12 引入了有状态滑动窗口聚合；两者均运行在紧凑的逻辑 DAG 上，源速率固定为 0.8 M/s。数据同步（DS）反映了一个对生产至关重要的摄取流水线，该流水线将记录从消息队列传输到 Apache Hive，建模为一个双节点 DAG，输入速率在 1 M/s 至 6.4 M/s 之间变化。样本拼接（SS）代表一种更复杂的工作负载，其特点是包含数十个逻辑节点和双输入流，源速率在 5 K/s 至 45 K/s 之间波动。

B. 作业启动开销比较

我们使用具有代表性的复杂查询 Nexmark Q12，评估 StreamShield 加速作业启动的效果。实验通过增加 TaskManager（TM）的数量进行扩展，同时将每个 TM 的任务槽数固定为两个。我们从作业启动开销的角度比较基线系统与 StreamShield，并将结果分解为三个阶段：作业解析、资源分配和任务部署。在本次评估中，我们将实验集群中的 SSD 存储替换为基于 HDD 的存储，以模拟真实在线环境中的性能瓶颈。

图 5 比较了基线和 StreamShield 在不同集群规模下的作业启动开销。在 512 个任务管理器下（图 5a），两个系统的解析开销都很小（315 毫秒 vs. 427 毫秒），但资源分配是启动成本的主要部分：基线需要 234,977 毫秒，而 StreamShield 需要 158,613 毫秒。任务部署呈现出类似趋势，StreamShield 略优于基线（12,100 毫秒 vs. 9,446 毫秒）。在 1024 个任务管理器下（图 5b），解析开销仍然较小（268 毫秒 vs. 240 毫秒），而基线的资源分配开销升至 338,615 毫秒，是 StreamShield（142,767 毫秒）的两倍多。部署开销也出现差异：基线达到 15,565 毫秒，而 StreamShield 为 8,272 毫秒。在最大的规模——2048 个任务管理器下（图 5c），基线遭遇严重的资源分配延迟（500,889 毫秒），而 StreamShield 将这一开销降至 201,234 毫秒。两种情况下的解析开销都不大（800 毫秒 vs. 438 毫秒），而 StreamShield 的任务部署开销再次更低（30,947 毫秒 vs. 35,476 毫秒）。

总体而言，结果表明，资源分配是造成作业启动开销的主要因素，而 StreamShield 显著缓解了这一瓶颈。随着集群规模扩大，StreamShield 将资源分配开销降低了 50% 以上，并提高了任务部署效率，从而显著加快了大规模场景下的作业初始化。



（a）512 个任务管理器　　（b）1024 个任务管理器　　（c）2048 个任务管理器

图 5：不同数量任务管理器下的作业启动开销对比。

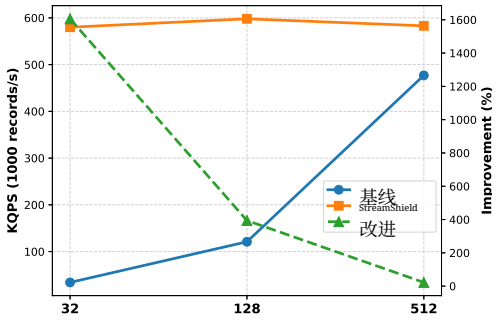


图 6：启用和未启用自适应 Shuffle 时的 QPS。

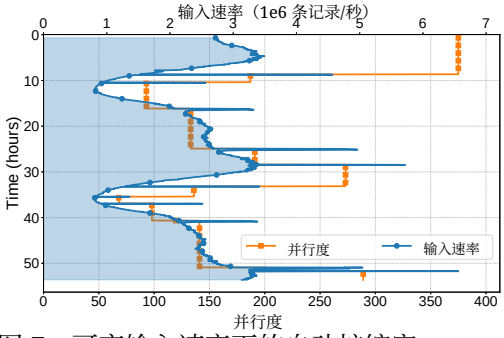


图 7：可变输入速率下的自动扩缩容。

C. 自适应 *Shuffle* 的效果

为评估自适应 Shuffle 的有效性，我们选择了包含两个逻辑算子的 Nexmark Q2。为引入工作负载不均衡，在处理每条输入记录之前，第二个算子中 10% 的任务会随机延迟 1000 毫秒。算子并行度固定为 8，我们将基线的吞吐量与启用了第三节 A 部分提出的基于积压的洗牌机制的 StreamShield 进行比较。如图 6 所示，在不同集群规模下，StreamShield 的表现始终优于基线。在 32 个任务管理器时，基线遭遇严重的数据倾斜，吞吐量仅为 34 KQPS，而 StreamShield 可稳定达到 580 KQPS，提升超过 1600%。在 128 个任务管理器时，基线达到 121 KQPS，而 StreamShield 达到 598 KQPS，提升约 400%。在 512 个任务管理器时，基线达到 477 KQPS，性能差距有所缩小，但 StreamShield 仍达到 583 KQPS。

StreamShield 采用基于积压的Shuffle 后能够保持相对稳定的 QPS，这得益于它能够在运行时动态平衡负载分布。在基线方案中，静态的轮询 Shuffle策略会导致数据分配不均，部分任务接收到的记录远多于其他任务，迅速成为瓶颈。这种不均衡会迫使整个流水线降速，在数据倾斜的负载下造成吞吐量急剧下降。相比之下，基于积压的Shuffle 会持续监控下游算子的积压状态，并将记录从拥堵的任务动态重新分配给负载较轻的任务，从而有效缓解慢任务并减少排队延迟。因此，StreamShield 避免了基线方案中出现的性能骤降，并能在不同的集群规模下保持稳定的吞吐量，展现了自适应 Shuffle 应对数据倾斜引发的工作负载不均衡的稳健性。

D. 自动扩缩容的效果

为评估 StreamShield 中所提出自动扩缩技术的有效性，我们选取数据同步（DS）作业作为代表性工作负载。在执行期间，输入速率持续变化，我们观察 StreamShield 如何通过调整作业内某个逻辑算子的并行度来适应变化。图 7 展示了 StreamShield 在波动工作负载下自动扩缩容的动态行为。在 55 小时的执行期间，输入速率（蓝色曲线，顶部坐标轴）变化显著，在每秒 100 万到接近 700 万条记录之间波动。开始时，输入速率稳步增长，StreamShield 通过将并行度从约 150 提高到 200 作出响应，从而确保足够的处理能力。大约在第 15 小时，输入速率急剧下降，并行度也相应降低，从而避免不必要的资源占用。在 20 至 35 小时之间也出现了类似模式：输入速率的突增触发并行度快速扩展至接近 200，随后在工作负载减弱时缩减并行度。在 40 至 55 小时之间，系统再次面临输入速率接近 7M/s 的激增；作为响应，并行度提高到接近 300，体现了系统弹性提供大规模资源的能力。达到峰值后，随着输入速率回落，并行度逐步降低，凸显了需求减弱后及时释放资源的机制效率。

总体而言，该图表明，StreamShield 的自动扩缩技术能够维持输入速率与

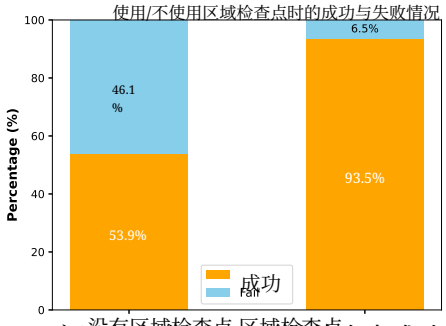


图 8：启用和未启用 RC 时的检查点成功率。

分配的并行度之间的强相关性。系统会通过提高并行度，及时响应工作负载突发，以保持吞吐量；当输入速率下降时，则会降低并行度以节省资源。这种自适应调整不仅能缓解重负载下的性能瓶颈，还能避免轻负载时资源过度配置，从而提升长期运行的流处理作业的效率与弹性。

E. 区域检查点的效果

为评估区域检查点（RC）的有效性，我们采用了 DS 作业这一代表性工作负载；该工作负载约占我们内部流处理应用的 10%。确定 RC 是否能为这类具有代表性的作业带来切实收益至关重要。为模拟真实的不稳定情况，我们在任务执行期间随机注入 5% 概率的 HDFS 上传变慢故障，同时将检查点间隔配置为 30 秒。每项实验运行 12 小时，以确保系统经历足够多的恢复事件，从而能够对启用和未启用 RC 的配置进行受控且可复现的比较。该设置确保了受控且可复现的评估环境，并使我们能够直接评估 RC 在分布式环境中提升 Flink 检查点可靠性方面的收益。

图 8 报告了 DS 工作负载在启用和未启用 RC 时的检查点成功率。未启用 RC 时，仅有 53.9% 的检查点成功完成（2326 个中有 1253 个），而 46.1% 的检查点因模拟的 HDFS 上传变慢而失败。相比之下，启用 RC 后可靠性显著提升，93.5% 的检查点成功完成（2244 个中有 2099 个），仅有 6.5% 失败。这一显著差异凸显了 RC 在缓解部分故障的级联效应方面的稳健性。通过隔离故障域并在区域级别重试，RC 避免了全局检查点中止，并维持了高得多的完成率。这些结果证实，RC 在提升大规模数据同步作业的检查点可靠性方面非常有效，而此类作业对频繁的状态持久化较为敏感。

F. 单任务恢复的效果

在评估单任务恢复时，我们选择了 SS 作业。该作业需要实时结果交付，同时只能容忍少量数据丢失。它在视频推荐系统中执行双流连接，将用于生成推荐结果的特征数据

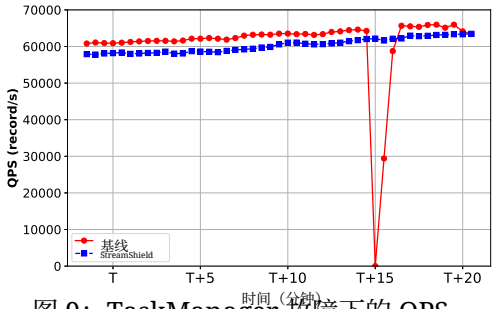


图 9：TaskManager 故障下的 QPS。

与推荐后行为反馈（例如点击事件或观看时长）相结合。

为评估故障的影响，我们在执行过程中主动终止一个 TaskManager，并监测两种设置下作业的吞吐量（QPS）：基线和采用单任务恢复的 StreamShield，如图 9 所示。在基线设置中，大约在 T+15 分钟时注入的故障会迫使整个作业重启，导致吞吐量完全降至零，随后进入持续数分钟的逐步恢复过程，在此期间服务始终不可用。这是因为拼接工作负载涉及一个 Join算子，其上游任务与下游任务之间采用全对全数据交换模式；原生故障转移机制会将单个任务故障传播至所有相连任务，从而触发区域级恢复。对于实时系统而言，这种行为十分不利，因为即使是短暂的中断，也可能显著降低用户体验。

相比之下，StreamShield 支持单任务恢复，并能隔离失败的子任务，而无需触发全局重启。因此，在故障事件期间，作业的吞吐量几乎保持稳定，仅在 T+15 附近出现轻微且短暂的波动。这种细粒度恢复机制无需重新初始化未受影响的任务，从而大幅缩短恢复时间并减轻性能中断。总的来说，该实验表明，单任务恢复可为实时流处理工作负载提供强大的容错能力，即使发生 TaskManager 故障，也能确保服务持续可用。

七、结论

本文介绍了 StreamShield，这是一个构建于字节跳动全球最大规模之一的 Apache Flink 集群之上的弹性框架，并从引擎、集群和发布三个角度进行了精心设计。StreamShield 集成了细粒度容错、运行时优化、混合复制和依赖感知机制，以应对执行层和外部系统中的故障。此外，在部署前，严格的测试与发布流水线会在易发生故障和高负载场景下验证这些机制，进一步增强生产环境弹性。

未来工作将聚焦于主动诊断与恢复，利用机器学习调整算子并行度，并在异常事件影响业务关键型应用之前将其检测出来。

参考文献

[1] P. Carbone、A. Katsifodimos、S. Ewen、V. Markl、S. Haridi与K. Tzoumas，“Apache Flink：单一引擎中的流处理与批处理”，《数据工程技术委员会通报》，第38卷，第4期，2015。[2] B. Chandramouli与J. Goldstein，“Shrink：为流式处理制定韧性解决方案”，VLDB 论文集，第10卷，第5期，第505–516页，2017。[3] J.-H. Hwang、M. Balazinska、A. Rasin、U. Cetintemel、M. Stonebraker与S. Zdonik，“分布式流处理的高可用算法”，ICDE。IEEE，2005，第779–790页。[4] Y. Xing、J.-H. Hwang、U. Cetintemel与S. Zdonik，“为分布式流处理中的负载变化提供韧性”，VLDB，第6卷，2006，第775–786页。[5] Y. Mei、L. Cheng、V. Talwar、M. Y. Levin、G. Jacques-Silva、N. Simha、A. Banerjee、B. Smith、T. Williamson、S. Yilmaz等，“Turbine：Facebook面向流处理的服务管理平台”，ICDE，2020，第1591–1602页。[6] “Apache Hadoop HDFS”，https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html，访问日期：2025/10/28]。[7] P. Hunt、M. Konar、F. P. Junqueira与B. Reed，“ZooKeeper：面向互联网规模系统的无等待协调”，USENIX年度技术会议，2010。[8] W. Xiang、Y. Li、Y. Ren、F. Jiang、C. Xin、V. Gupta、C. Xiang、X. Song、M. Liu、B. Li等，“G¨odel：字节跳动的统一大规模资源管理与调度”，2023年ACM 云计算研讨会论文集，2023，第308–323页。[9] C. Rosenthal等，《混沌工程》，第1版。O’Reilly Media, Inc.，2020。[10] P. C. Shekhar，“混沌测试：分布式架构中系统韧性的主动框架”，2024。[11] T. Akidau、R. Bradshaw、C. Chambers、S. Chernyak、R. J. Fern´andez- Moctezuma、R. Lax、S. McVeety、D. Mills、F. Perry、E. Schmidt等，“数据流模型：在大规模、无界、乱序数据处理中平衡正确性、延迟与成本的实用方法”，VLDB 论文集，第8卷，第12期，第1792–1803页，2015。[12] D. G. Murray、F. McSherry、R. Isaacs、M. Isard、P. Barham与M. Abadi，“Naiad：一个及时数据流系统”，第24届ACM操作系统原理研讨会论文集，2013，第439–455页。[13] P. Carbone、G. F´ora、S. Ewen、S. Haridi与K. Tzoumas，“面向分布式数据流的轻量级异步快照”，arXiv预印本 arXiv:1506.08603，2015。[14] “Apache Flink文档：检查点”，https://nightlies.apache.org/flink/flink-docs-master/docs/ops/state/checkpoints/，2025，访问日期：2025-09-28。[15] A. Flink，“Flink文档：保存点”，https://nightlies.apache.org/flink/flink-docs-master/docs/ops/state/savepoints/，2025，访问日期：2025-09-28。[16] N. Rivetti、E. Anceaume、Y. Busnel、L. Querzoni与B. Sericola，“分布式流处理系统中洗牌分组的在线调度”，第17届国际Middleware会议论文集，2016，第1–12页。[17] N. Kruber.（2019年6月）深入剖析Flink网络栈。https: //flink.apache.org/2019/06/05/a-deep-dive-into-flinks-network-stack/。[18] R. Kang、Y. Chen、Y. Liu、F. Jiang、Q. Li、M. Ma、J. Liu、G. Zhao、T. Zhang、J. Chen与L. Zhang，“ABase：面向大规模云环境中多样化动态工作负载的多租户NoSQL无服务器数据库”，2025年国际数据管理会议配套论文集，2025，第471–484页。[19] A. Floratou、A. Agrawal、B. Graham、S. Rao与K. Ramasamy，“Dhalion：Heron中的自调节流处理”，VLDB 论文集，第10卷，第12期，第1825–1836页，2017。[20] R. Castro Fernandez、M. Migliavacca、E. Kalyvianaki与P. Pietzuch，“利用算子状态管理在流处理中整合横向扩展与容错”，SIGMOD，2013，第725–736页。[21] L. Xu、B. Peng与I. Gupta，“Stela：使流处理系统能够按需缩容与横向扩展”，2016年IEEE国际云工程会议（IC2E）。IEEE，2016，第22–31页。[22] V. Kalavri、J. Liagouris、M. Hoffmann、D. Dimitrova、M. Forshaw与T. Roscoe，“只需三步：面向分布式流式数据流的快速、准确的自动扩缩容决策”，第13届USENIX操作系统设计与实现研讨会（OSDI），2018 年，第 783–798 页。[23] T. Z. Fu、J. Ding、R. T. Ma、M. Winslett、Y. Yang 和 Z. Zhang，《DRS：实时流分析的自动伸缩》，IEEE/ACM 网络汇刊，第 25 卷，第 6 期，第 3338–3352 页，2017 年。[24] B. Lohrmann、P. Janacik 和 O. Kao，《具有延迟保证的弹性流处理》，载于《2015 年 IEEE 第35届分布式计算系统国际会议》，IEEE，2015 年，第 399–410 页。[25] Y. Mao、Z. Chen、Y. Zhang、M. Wang、Y. Fang、G. Zhang、R. Shi 和 R. T. Ma，《StreamOps：字节跳动流式服务的云原生运行时管理》，VLDB 论文集，第 16 卷，第 12 期，第 3501–3514 页，2023 年。[26] Y. Han、L. Chen、H. Wang、Z. Chen、Y. Zhang、C. Yang、K. Hao 和 Z. Yang，《从过去中学习：流处理系统的自适应并行度调优》，载于 ICDE，2025 年，第 3535–3548 页。[27] K. Xu、W. Hu、J. Leskovec 和 S. Jegelka，《图神经网络的能力有多强？》，载于第七届国际学习表征会议（ICLR 2019）。OpenReview.net，2019 年。[28] W. L. Hamilton、Z. Ying 和 J. Leskovec，《大型图上的归纳式表示学习》，载于 NIPS 2017，I. Guyon、U. von Luxburg、S. Bengio、H. M. Wallach、R. Fergus、S. V. N. Vishwanathan 和 R. Garnett 编，2017 年，第 1024–1034 页。[29] K. Xu、C. Li、Y. Tian、T. Sonobe、K.-i. Kawarabayashi 和 S. Jegelka，《使用跳跃知识网络进行图表示学习》，载于 ICML，2018 年，第 5453–5462 页。[30] Y. Mao、R. Yin、L. Lei、P. Ye、S. Zou、S. Tang、Y. Guo、Y. Yuan、X. Yu、B. Wan 等，《ByteMQ：字节跳动的云原生流式数据层》，载于《2024 年 ACM 云计算研讨会论文集》，2024 年，第 774–791 页。[31] Apache Flink 项目，《任务故障恢复》，https://nightlies.apache.org/flink/flink-docs-master/docs/ops/state/task failure recovery/，2025 年。[32] S. Dong、A. Kryczka、Y. Jin 和 M. Stumm，《Rocksdb：为大规模应用提供服务的键值存储，其开发优先级的演变》，ACM 存储汇刊（TOS），第 17 卷，第 32 期，2021 年 4 月，第 1–页。[33] 《Nexmark 基准测试》，https://beam.apache.org/documentation/sdks/java/testing/nexmark/，2024, [访问日期：2025/10/28]。[34] M. Balazinska、H. Balakrishnan、S. R. Madden 和 M. Stonebraker，《Borealis 分布式流处理系统中的容错》，ACM 数据库系统汇刊，第 33 卷，第 1 期，第 1–44 页，2008 年。[35] M. A. Shah、J. M. Hellerstein 和 E. Brewer，《高可用、容错的并行数据流》，载于《2004 年 ACM SIGMOD 数据管理国际会议论文集》，2004 年，第 827–838 页。[36] Y. Gu、Z. Zhang、F. Ye、H. Yang、M. Kim、H. Lei 和 Z. Liu，《流处理系统中高可用性的实证研究》，载于 Middleware（伴随会议），2009 年，第 23 页。[37] J.-H. Hwang、Y. Xing、U. Cetintemel 和 S. Zdonik，《用于流处理的协作式自配置高可用性解决方案》，载于 ICDE，IEEE，2007 年，第 176–185 页。[38] Y. Kwon、M. Balazinska 和 A. Greenberg，《使用分布式复制文件系统实现容错流处理》，VLDB 论文集，第 1 卷，第 1 期，第 574–585 页，2008 年。[39] Z. Zhang、Y. Gu、F. Ye、H. Yang、M. Kim、H. Lei 和 Z. Liu，《流处理系统高可用性的混合方法》，载于《2010 年 IEEE 第30届分布式计算系统国际会议》，IEEE，2010 年，第 138–148 页。[40] T. Heinze、M. Zia、R. Krahn、Z. Jerzak 和 C. Fetzer，《弹性数据流处理系统的自适应复制方案》，载于《第九届 ACM 分布式事件系统国际会议论文集》，2015 年，第 150–161 页。[41] L. Su 和 Y. Zhou，《大规模并行流处理引擎中对相关故障的容忍》，载于《2016 年 IEEE 第32届数据工程国际会议（ICDE）》，IEEE，2016 年，第 517–528 页。[42] D. Borthakur，《Hadoop 分布式文件系统：架构与设计》，Hadoop 项目网站，第 11 卷，第 2007 期，第 21 页，2007 年。[43] S. Malhotra、F. Yashu、M. Saqib、D. Mehta、J. Jangid 和 S. Dixit，《评估分布式文件系统中的容错性和可扩展性：GFS、HDFS 和 MinIO 案例研究》，arXiv 预印本 arXiv:2502 01981，2025 年。[44] H. Li、C. Lu 和 C. D. Gill，《RT-Zookeeper：降低协调服务的恢复延迟》，ACM 嵌入式计算系统汇刊（TECS），第 20 卷，第 5s 期，第 1–22 页，2021 年。

[45] Y. Wang、L. Huang、Z. Wang、V. Kalavri 和 I. Matta，《CAPSys：面向数据流处理的竞争感知任务放置》，载于第二十届欧洲计算机系统会议论文集，2025 年，第 654–670 页。[46] B.-J. Kwak、N.-O. Song 和 L. E. Miller，《性能分析指数退避”，*IEEE/ACM* 网络汇刊，第 13 卷第 2 期，第 343–355 页，2005 年。[47] G. Ramalingam 和 K. Vaswani，“通过幂等性实现容错”，载于 第 *40* 届 *ACM SIGPLAN-SIGACT*编程语言原理研讨会论文集，2013 年，第 249–262 页。