Parallel Computing in the Cloud: Scheduling, Elasticity, and Resilience in Heterogeneous Environments

Andrew Photinakis
Rochester Institute of Technology
acp7795@rit.edu

***Abstract*—Cloud computing has undergone a fundamental paradigm shift from static, homogeneous clusters to dynamic, heterogeneous environments that demand fine-grained elasticity and deep resilience. This paper explores the evolution of parallel scheduling by addressing the limitations of traditional models like MapReduce, which struggle with the I/O contention and hardware heterogeneity inherent in multi-tenant clouds. It analyzes three distinct systems: MRS for optimizing MapReduce scheduling through heuristic I/O modeling, Dandelion for unlocking true serverless elasticity by redefining the execution interface, and ElasWave for enabling resilient, fault-tolerant Large Language Model (LLM) training at scale. By synthesizing these approaches through the lens of the cloud-native lifecycle, this paper demonstrates that future systems must abandon legacy POSIX interfaces in favor of application-aware, "elastic-native" designs to minimize resource waste and recovery latency.**

***Index Terms*—Cloud Computing, Parallel Scheduling, Serverless, Resilience, Hybrid Parallelism**

I. Introduction

The architecture of parallel computing has historically relied on the assumption of hardware homogeneity. Traditional clusters were built with identical processors, predictable network latencies, and dedicated storage, allowing schedulers to operate with simple heuristics. However, the migration of high-performance computing to the cloud has fundamentally invalidated these assumptions. Modern cloud environments are defined by dynamic heterogeneity, multi-tenant interference, and frequent hardware failures, necessitating a complete rethinking of how parallel tasks are scheduled, scaled, and recovered [1].

A. The Shift to Cloud-Native Parallelism

The evolution of cloud computing can be categorized into two distinct eras. "Cloud 1.0" was characterized by the delivery of infrastructure resources (IaaS), where users treated the cloud as a remote data center for running virtual machines (VMs). In this model, legacy monolithic applications were often "lifted and shifted" without modification, relying on heavy operating systems and standard POSIX interfaces. While this provided utility, it failed to exploit the unique properties of the cloud environment [1].

"Cloud 2.0," or Cloud-Native Computing, represents a paradigm shift where applications are architected specifically for elasticity and distribution. This era is defined by the decomposition of monoliths into microservices, the adoption of lightweight containers over heavy VMs, and the use of automated orchestration platforms. However, this shift introduces significant complexity. Applications now consist of thousands of interdependent services, creating complex Directed Acyclic Graphs (DAGs) rather than simple linear workflows. Furthermore, the underlying hardware is no longer dedicated; it is shared among multiple tenants, leading to unpredictable performance variability [2].

B. The Core Challenges

Three primary challenges have emerged in this new landscape: heterogeneity-aware scheduling, true elasticity, and resilience at scale.

Scheduling in the cloud is complicated by the "noisy neighbor" effect. When multiple VMs share the same physical host, they compete for shared resources such as disk I/O and memory bandwidth. Traditional schedulers, such as First-In-First-Out (FIFO) or Fair Schedulers, ignore these external factors, assuming that task execution time is a function solely of CPU speed. This leads to suboptimal placement, where critical tasks are assigned to contended nodes, creating stragglers that bottle-neck the entire parallel job [3].

The promise of elasticity, which is the ability to scale resources perfectly to match demand, remains largely unfulfilled. Current serverless (Function-as-a-Service) platforms claim to scale to zero, but in practice, they suffer from high cold-start latencies caused by the overhead of booting guest operating systems and initializing runtimes. To mask this latency, providers pre-provision idle "warm" sandboxes, wasting vast amounts of memory and breaking the economic promise of paying only for active usage [2].

The massive scale of modern Artificial Intelligence (AI) workloads has made resilience a critical concern. Training Large Language Models (LLMs) requires thousands of accelerators running for weeks. At this scale, hardware failure is not an exception but a routine occurrence. Traditional "fail-stop" faults (crashes) and modern "fail-slow" faults (performance degradation) disrupt training frequently. Relying on checkpoint-restart mechanisms, which require stopping the entire cluster and reloading data from disk, results in unacceptable downtime and wasted computation [4].

C. Roadmap

This paper analyzes three state-of-the-art systems that address these challenges across the cloud-native lifecycle. We examine the **MRS** algorithm, which introduces a heuristic for scheduling MapReduce tasks in the presence of I/O contention. We explore **Dandelion**, a serverless platform that redefines the execution interface to achieve microsecond-scale elasticity. Finally, we analyze **ElasWave**, a system designed for resilient hybrid-parallel training that recovers from failures without full restarts. By contrasting these approaches with related works such as Quasar, Firecracker, Oobleck, and Pollux, we identify the trajectory of future research in cloud parallelization.

II. Literature Review

The literature on cloud parallelization reveals a progression from hardware-centric optimizations to application-aware designs. This section contrasts the approaches of the primary focus papers against related state-of-the-art solutions to highlight their unique contributions and trade-offs.

A. The Cloud-Native Lifecycle Framework

To contextualize the contributions of the analyzed systems, it is necessary to adopt a structured view of the cloud-native service lifecycle. Deng et al. [1] decouple this lifecycle into four distinct states, each presenting unique parallelization challenges:

1) **Building:** This state involves the development and packaging of applications. The shift from monolithic codebases to microservices requires new build artifacts. While traditional virtualization relied on heavy disk images, modern approaches utilize lightweight container images or, as proposed by Dandelion, serverless functions compiled into intermediate representations like WebAssembly.
2) **Orchestration:** This state focuses on the scheduling and placement of tasks. The challenge here, addressed by MRS, is mapping abstract tasks to physical resources while accounting for heterogeneity and interference. Effective orchestration must maximize cluster utilization while adhering to Quality of Service (QoS) constraints.
3) **Operation:** This state covers the runtime management of services, specifically auto-scaling. The transition from manual provisioning to automated elasticity is the defining characteristic of Cloud 2.0. Systems like Dandelion aim to push this to its theoretical limit by enabling per-request scaling.
4) **Maintenance:** The final state deals with fault tolerance and anomaly detection. As systems scale, maintenance shifts from reactive repair to proactive resilience. ElasWave operates in this domain, ensuring that the inevitable hardware failures of the maintenance phase do not disrupt the continuous execution of the application [1].

B. Scheduling: Analytical Heuristics vs. Machine Learning

The problem of scheduling in heterogeneous environments is often framed as minimizing task completion time under uncertainty. The MRS (MapReduce Scheduler) approach addresses this via a "white-box" analytical model. Ling et al. derive a specific mathematical formula to explicitly model the relationship between a host's load level and the resulting I/O interference. This allows the scheduler to make deterministic decisions based on observable metrics [3].

In contrast, **Quasar**, presented by Delimitrou and Kozyrakis, employs a "black-box" data-driven approach. Quasar utilizes Collaborative Filtering, a machine learning technique, to classify incoming workloads based on their similarity to previously seen jobs. By inferring performance characteristics rather than modeling them explicitly, Quasar achieves broader generalization across different workload types (e.g., batch vs. latency-sensitive) [5]. However, this approach relies heavily on the quality and quantity of training data. MRS, while less generalizable, offers stronger theoretical guarantees (a 7-approximation) for the specific domain of MapReduce, making it potentially more robust for strictly defined batch processing tasks.

C. Elasticity: Virtualization vs. Language-Based Isolation

The industry standard for serverless isolation is represented by Firecracker, developed by AWS. Firecracker utilizes lightweight MicroVMs that strip down the KVM hypervisor to the bare essentials. This architecture prioritizes compatibility; by providing a Linux kernel, it allows legacy POSIX applications to run unmodified. However, this compatibility comes at the cost of performance, with boot times hovering around 125ms [6].

Dandelion takes a radically different approach by rejecting the POSIX interface entirely. It enforces a strict separation between "pure compute" functions and platform-managed I/O. This constraint allows Dandelion to eschew hardware virtualization in favor of language-based isolation or hardware capabilities like CHERI. The result is a system that can boot a sandbox in microseconds rather than milliseconds. The trade-off is clear: Firecracker optimizes for the backward compatibility required by "lift-and-shift" strategies, while Dandelion optimizes for the performance requirements of true cloud-native elasticity [2].

D. Resilience: Static Templates vs. Dynamic Reshaping

Resilience in AI training has evolved from simple checkpointing to sophisticated in-memory recovery. Oobleck addresses fault tolerance by introducing pre-computed "Pipeline Templates." Oobleck adopts a "planning-execution co-design" approach. During the planning phase, it pre-generates a set of heterogeneous pipeline templates and instantiates $(f+1)$ logically equivalent pipeline replicas to tolerate f simultaneous failures. This guarantees that at least one valid configuration always exists to cover the remaining resources. While this provides a provable guarantee of fault tolerance, it requires holding spare capacity in reserve (the $+1$ replica), which can be economically inefficient [7].

ElasWave, conversely, operates without explicit spare replicas. By implementing dynamic, in-place reshaping of the computation graph, ElasWave allows training to continue on

$N-1$ nodes immediately after a failure. Rather than relying on static templates, ElasWave's scheduler actively migrates model layers and resizes batches to fit the remaining resources. This utilizes 100% of the available hardware at all times, making it more cost-efficient for massive clusters where holding an entire pipeline replica in reserve would be prohibitively expensive. Thus, Oobleck optimizes for recovery speed via redundancy, while ElasWave optimizes for hardware efficiency via elasticity [4]. Complementing these is Pollux, which focuses on "Goodput" (statistical efficiency), suggesting that future systems should combine ElasWave's failure recovery with Pollux's optimization logic [8].

III. Parallelization Methods, Algorithms, and Architectures

This section details the structural designs and algorithmic foundations of the three focus systems, explaining how they organize parallel execution to address their respective challenges.

A. MRS: Geometric Interval Scheduling for MapReduce

The standard MapReduce architecture utilizes a master node to coordinate a set of worker nodes, dividing tasks into Map and Reduce phases. A critical constraint is the precedence requirement: Reduce tasks cannot commence until their corresponding Map tasks are complete. In a homogeneous cluster, this is trivial. In a heterogeneous cloud, stragglers can stall the entire Reduce phase.

To mitigate this, MRS introduces a scheduling architecture based on **Geometric Time Intervals**. The timeline is discretized into a sequence of intervals defined by $t_n = 2^{n-1}$ (e.g., [1, 2], [2, 4], [4, 8]). This discretization simplifies the continuous scheduling problem into discrete windows. Within each interval, MRS employs Graham's List Scheduling, a classical algorithm where tasks are ordered by priority and assigned to the next available processor. By combining geometric intervals with list scheduling, MRS enforces precedence constraints while dynamically adapting to the varying processing capacities of heterogeneous nodes. The scheduler constantly re-evaluates task placement based on the interference model, effectively steering critical tasks away from heavily contended nodes [3].

B. Dandelion: The Coordinator-Engine Dataflow Model

Dandelion abandons the monolithic process model in favor of a dataflow architecture. Applications are defined as Directed Acyclic Graphs (DAGs) of functions. The execution runtime is architected around a central **Coordinator** and two specialized types of execution engines:

1) **Compute Engines:** These execute untrusted user code ("pure compute" functions). To maximize CPU efficiency, compute engines operate on a run-to-completion model. A single CPU core is dedicated to a compute task until it finishes, eliminating the overhead of context switching.
2) **Communication Engines:** These are trusted, platform-managed components that handle all I/O operations. Unlike compute engines, they utilize cooperative multi-threading to handle thousands of concurrent requests on a single core.

To facilitate high-performance communication between these decoupled engines without incurring kernel overhead, Dandelion utilizes a shared-memory interconnect. In traditional architectures, passing data between isolated contexts requires system calls (e.g., `pipe` or `socket`), which force a context switch. Dandelion replaces this with unidirectional shared memory channels (SHM). When a Communication Engine receives an external request, it writes the payload directly into the SHM region of the target Compute Engine and triggers a lightweight user-space notification. This "zero-copy" architecture ensures that data flows through the system purely in user space, avoiding the expensive boundary crossings that define the performance floor of systems like Firecracker [2].

C. ElasWave: Multi-Dimensional Hybrid Parallelism

ElasWave is designed for the complex requirements of Hybrid Parallelism, which is necessary for training models that exceed the memory capacity of a single GPU. Hybrid parallelism combines Data Parallelism (splitting the dataset), Pipeline Parallelism (splitting the model layers), and Tensor Parallelism (splitting individual matrix operations).

ElasWave's architecture features a Multi-Dimensional Scheduler that coordinates three distinct aspects of the system simultaneously:

- **Dataflow Dimension:** The scheduler can dynamically resize micro-batches. If a node fails and the effective cluster size shrinks, the scheduler adjusts the micro-batch size to preserve the global batch size, ensuring that mathematical convergence properties remain stable.
- **Graph Dimension:** The scheduler manages the partitioning of the neural network graph. It can migrate model layers between pipeline stages to rebalance the computational load when the number of available accelerators changes.
- **Hardware Dimension:** The scheduler interfaces with hardware controls to apply Dynamic Voltage and Frequency Scaling (DVFS). This is used to eliminate residual performance "bubbles", which're idle times in the pipeline caused by imperfect partitioning, by speeding up or slowing down specific devices.

This holistic architectural view allows ElasWave to absorb failures and stragglers by reconfiguring the software pipeline around the hardware issues [4].

The integration of the Hardware Dimension is particularly novel. In a pipelined training setup, "bubbles" are inevitable due to imperfect graph partitioning or transient hardware fluctuations. Traditional schedulers leave these bubbles as wasted time. ElasWave's scheduler identifies these bubbles proactively and applies DVFS to mitigate their impact.

If a specific stage is identified as a straggler (running slower than others), the scheduler boosts its frequency to close the gap. Conversely, if a stage is consistently waiting (creating bubbles), the scheduler may lower its frequency to save energy without impacting the overall time-to-solution.

This fine-grained control allows ElasWave to harmonize the software pipeline with the physical reality of the hardware, treating frequency as a mutable resource rather than a static constraint [4].

1) Mechanism: Interleaved ZeRO Partitions: A critical barrier to elasticity in large-scale training is the movement of model states. When a pipeline is resized, parameters must be transferred between nodes. ElasWave mitigates the latency of this transfer through **Asynchronous Parameter Migration** combined with interleaved ZeRO partitions.

In standard ZeRO (Zero Redundancy Optimizer) implementations, parameters are sharded across all GPUs to save memory. ElasWave interleaves these partitions such that the data required for the immediate next step of computation is prioritized. The scheduler identifies the minimal subset of parameters needed to restart the pipeline on the new topology and transfers them first. The remaining parameters are migrated asynchronously in the background while computation resumes. This overlapping of communication and computation effectively hides the cost of state migration, allowing the system to maintain high GPU utilization even during active topological reconfiguration [4].

IV. Research Advances, Developments, and Applications

This section details the specific technical contributions, quantitative results, and mechanisms proposed in the focus papers.

A. MRS: Modeling I/O Interference

The primary research advance of the MRS paper is the development of a rigorous mathematical model for quantifying I/O interference. In virtualized environments, the processing time of a task is non-linear with respect to the host's load. Ling et al. propose the following interference model:

$$p_{uk} = p'_u \times \epsilon \left(\frac{a}{l_k} + b \right) \tag{1}$$

In this equation, p_{uk} is the estimated processing time of task u on processor k. p'_u represents the task's ideal processing time in isolation. The term $\epsilon(\frac{a}{l_k} + b)$ is the impact factor, where l_k is the current load level of the host (measured by the number of active VMs or I/O operations), and a and b are empirical coefficients derived from regression analysis of the specific hardware environment.

The MRS algorithm implements this interference model through a three-step heuristic designed to approximate the optimal schedule:

1) **Task Prioritization:** First, all tasks in the job DAG are sorted based on their release times and deadline constraints. MRS specifically prioritizes tasks on the "critical path"—the longest sequence of dependent tasks (Map $\rightarrow$ Shuffle $\rightarrow$ Reduce)—to ensure that delays in these tasks do not cascade through the system.
2) **Interval Assignment:** The algorithm iterates through the geometric intervals (t_n). For each interval, it identifies the subset of tasks that are ready to execute. This discretization reduces the search space, transforming the continuous scheduling problem into a series of smaller, tractable sub-problems.
3) **Interference-Aware Mapping:** In the final assignment step, MRS attempts to map high-priority tasks to nodes with the lowest load factor (l_k). Crucially, if the estimated completion time p_{uk} (calculated via the interference formula) on a specific node exceeds the current interval boundary, the task is deferred. This prevents the scheduler from assigning critical tasks to "noisy" neighbors that would cause them to violate the interval constraints, effectively quarantining stragglers before they occur [3].

By integrating this model into their heuristic, the authors proved that MRS achieves a **7-approximation** ratio for minimizing the Total Weighted Task Completion Time (TWTCT). In experimental evaluations on Amazon EC2, comparing against standard FIFO and Shortest-Task-First (STF) strategies, MRS demonstrated superior performance. Specifically, in worst-case scenarios involving heavy I/O contention, MRS achieved a competitive ratio of **2.13**, significantly outperforming FIFO (4.17) and STF (3.51). This validates the hypothesis that explicit interference modeling is essential for efficient cloud scheduling [3].

B. Dandelion: Achieving Microsecond Cold Starts

Dandelion's research contribution lies in demonstrating that the high latency of serverless cold starts is not an inherent property of the cloud, but a consequence of the POSIX interface. By utilizing CHERI hardware capabilities for memory safety and isolation, Dandelion eliminates the need for virtual memory page tables and kernel context switches during sandbox initialization.

The quantitative results are dramatic. Dandelion achieves a cold start time of approximately **89 μs**. This is several orders of magnitude faster than the 125ms required for Firecracker MicroVMs or the several seconds required for standard Docker containers. This speed enables a new operational model: "True Elasticity," where a fresh sandbox is booted for every single request. In trace-driven simulations using Azure Functions workloads, Dandelion reduced committed memory usage by **96%** compared to Knative autoscaling. Knative typically keeps containers "warm" for 10–20 minutes to avoid cold starts, wasting vast resources. Dandelion's ability to boot on-demand eliminates this waste entirely [2].

C. ElasWave: Dynamic Communicators and Consistency

ElasWave advances the state of the art in distributed training resilience through two novel mechanisms designed to minimize Mean Time To Recovery (MTTR) and ensure mathematical correctness.

ElasWave introduces the **Dynamic Communicator**. In standard distributed training frameworks like PyTorch, a node failure necessitates the teardown and reconstruction of the global communication group (using libraries like NCCL). This process typically takes 12–16 seconds, during which the entire cluster is idle. ElasWave's Dynamic Communicator edits the

group membership in-place, reusing existing connections and only establishing new links where necessary. This reduces communicator recovery time to 0.15–0.37 seconds, an acceleration of up to 82x compared to full rebuilds.

Second, ElasWave addresses the problem of Computation Consistency. When the number of workers changes, the distribution of data samples changes. This can alter the state of pseudo-random number generators (RNGs) used for operations like Dropout, leading to divergent training results. ElasWave implements RNG Resharding, which dynamically redistributes RNG states alongside data partitions. This ensures that a training run recovered after a failure produces the exact same loss curve and convergence trajectory as a run that never failed. Combined, these mechanisms improve overall training throughput by 1.60x over industrial baselines like TorchFT and 1.35x over ReCycle [4].

V. Critique, Scope, and Discussion

While the systems presented offer significant advancements, they also present new trade-offs and limitations. This section critiques the assumptions, applicability, and complexity of these approaches in the context of the broader cloud-native landscape.

A. The Obsolescence of Batch Assumptions in MRS

The MRS algorithm is built upon a solid mathematical foundation, but its applicability to modern Cloud 2.0 workloads is questionable. The paper relies on the assumption that "tenants have adequate budgets" and that minimizing completion time is the sole objective. In the current era of FinOps and cost-aware computing, budget is often a primary constraint, not a given. Furthermore, MRS assumes that "every job is relatively independent." This simplifies the scheduling problem to a set of batch tasks. However, modern cloud-native applications are characterized by highly interdependent microservices forming complex call graphs. The interference model used by MRS, which looks primarily at host load, may not capture the complex cascading latency effects seen in microservice architectures. Therefore, while MRS is highly effective for legacy batch analytics (like Hadoop), it may be outdated for orchestrating modern, latency-sensitive service meshes.

B. The Cost of Breaking POSIX

Dandelion's rejection of the POSIX standard is simultaneously its greatest strength and its most significant barrier to adoption. By prohibiting system calls, Dandelion achieves unprecedented performance. However, this creates a massive software engineering hurdle. The vast majority of existing software libraries and tools depend on POSIX APIs for threading, networking, and file I/O. Adopting Dandelion requires developers to rewrite their applications to fit the split "compute/communication" model. Unlike Docker containers or Firecracker MicroVMs, which allow organizations to "lift and shift" legacy code into the cloud, Dandelion demands a "greenfield" approach. The paper suggests that future compilers might automatically decompose legacy code, but until such tools mature, Dandelion will likely remain a niche solution for specialized high-performance workloads rather than a general-purpose replacement for containers.

C. Complexity vs. Observability in AI Training

ElasWave demonstrates that dynamic recovery is superior to static checkpointing in terms of throughput. However, this performance comes at the cost of extreme system complexity. The system is performing complex state manipulations—resizing batches, migrating layers, and resharding RNG states—in real-time while the cluster is running. The Cloud-Native Survey identifies "Observability" as a paramount challenge in distributed systems. Debugging a "silent data corruption" in ElasWave is exponentially harder than debugging a standard static cluster. If the loss curve diverges slightly, tracing the error to a specific migrated layer or mis-synced RNG state is a daunting task. Without advanced observability tools that can visualize these dynamic state transitions, the operational risk of deploying ElasWave may outweigh its performance benefits for many organizations.

D. Security Trade-offs in Shared Infrastructure

There is a fundamental tension between the lightweight isolation used by Dandelion (and similar language-based approaches) and the heavy hardware virtualization used by Firecracker. Dandelion argues that banning system calls reduces the attack surface sufficiently. However, the Cloud-Native Survey notes that multi-tenant security risks are exacerbated in shared infrastructure. Shared-memory environments are historically prone to architectural side-channel attacks (e.g., Spectre, Meltdown) that exploit the physical properties of the CPU cache. Hardware virtualization (like KVM) provides a robust barrier against these attacks by flushing state and isolating memory pages at the hardware level. It remains an open question whether Dandelion's software-based or capability-based isolation can provide the same defense-in-depth as MicroVMs in hostile, public cloud environments where malicious tenants actively try to breach isolation.

E. The Challenge of Standardization and Lock-In

A recurrent theme in the Cloud-Native Survey is the risk of vendor lock-in due to non-standard interfaces. While the shift to Cloud 2.0 was driven by open standards like Docker and Kubernetes, systems like Dandelion threaten to re-introduce lock-in through proprietary execution models.

Dandelion's requirement for applications to adhere to a strict "compute/communication" separation creates a high degree of coupling between the application logic and the platform's specific API. Unlike a container, which is portable across any cloud provider supporting the OCI standard, a Dandelion function is effectively tethered to the Dandelion runtime. This raises significant strategic concerns for enterprises. Adopting such an "elastic-native" architecture may yield performance gains, but it reduces portability, making it difficult to migrate workloads between cloud providers or back to on-premise data centers. Future research must address how to standardize these dataflow interfaces to prevent a fragmentation of the serverless ecosystem [1], [2].

VI. Conclusion

The domain of parallel computing in the cloud has evolved from a focus on managing static hardware resources to a focus on enabling dynamic, elastic application behaviors. Early innovations like the MRS algorithm addressed the immediate reality of heterogeneous hardware and I/O contention in virtualized data centers. However, as the cloud has matured into a platform for hyperscale AI and serverless microservices, the challenges have shifted toward latency and resilience.

Systems like Dandelion demonstrate that achieving "true elasticity" requires a fundamental rethinking of the interface between the application and the cloud, moving away from legacy OS abstractions toward specialized dataflow models. Similarly, ElasWave proves that resilience at the scale of thousands of GPUs cannot rely on simple redundancy; it requires deep application awareness and the ability to dynamically reshape computation in real-time. Ultimately, the future of parallel computing lies in the co-design of application logic and infrastructure, creating "elastic-native" systems where failure handling and scaling are integral, programmable components of the application itself.

References

[1] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, "Cloud-native computing: A survey from the perspective of services," 2023. [Online]. Available: https://arxiv.org/abs/2306.14402

[2] T. Kuchler, P. Li, Y. Zhang, L. Cvetković, B. Goranov, T. Stocker, L. Thomm, S. Kalbermatter, T. Notter, A. Lattuada, and A. Klimovic, "Unlocking true elasticity for the cloud-native era with dandelion," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP '25. ACM, Oct. 2025, p. 944–961. [Online]. Available: http://dx.doi.org/10.1145/3731569.3764803

[3] X. Ling, J. Yang, D. Wang, and Y. Wang, "Cluster scheduler on heterogeneous cloud," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, 2015, pp. 772–777.

[4] X. Kang, G. Xiang, Y. Wang, H. Zhang, Y. Fang, Y. Zhou, Z. Tang, Y. Lv, E. Maman, M. Wasserman, A. Zameret, Z. Bian, S. Chen, Z. Yu, J. Wang, X. Wu, Y. Zheng, C. Tian, and X. Chu, "Elaswave: An elastic-native system for scalable hybrid-parallel training," 2025. [Online]. Available: https://arxiv.org/abs/2510.00606

[5] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *SIGPLAN Not.*, vol. 49, no. 4, p. 127–144, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2644865.2541941

[6] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[7] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury, "Oobleck: Resilient distributed training of large models using pipeline templates," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. ACM, Oct. 2023, p. 382–395. [Online]. Available: http://dx.doi.org/10.1145/3600006.3613152

[8] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing, "Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/qiao