Fault-Tolerant and Reproducible Input Processing for Machine Learning

Master Thesis

Dan Kluser supervised by Dan Graur and Prof. Ana Klimovic

ML Input Data Processing

- More data is a key factor in Deep Learning’s success
- Data is expensive
- Augmenting dataset needs compute intensive online processing
- Often pre-processing becomes a bottleneck
  - 20% of jobs spend 1/3 in input pipeline
  - 30% of total compute time for pre-processing

Alleviating Bottleneck by Scaling Out

- Industry agrees: Both Meta and Google scale out preprocessing using disaggregated service
- Right-size resources for training and pre-processing independently
- Exploit synergies between jobs by having a fleet-wide view within the service

Issues when Scaling Out

1) Fault-Tolerance

As more nodes are involved in a job failure probabilities become non-negligible

2) Reproducibility

Distributing the job introduces additional sources of non-determinism

(DRR is joint work with Zak; presented by him)

Issues when Scaling Out

This presentation

1) Fault-Tolerance

As more nodes are involved in a job failure probabilities become non-negligible

Motivation for Fault-Tolerance

- Only 35.9% ML jobs at Meta did not experience any fault [over course of a week]
- Use of transient cloud resources (i.e. SpotVMs)
  - At least 60% cheaper, but may be preempted at any time
- Live migration of nodes in datacenters

Requirements for Fault-Tolerant Processing

1. Disaggregated Distributed Pre-Processing
2. Correctness (Exactly-Once)
3. Performance (Bounded overhead; recover some progress)
4. Compatibility
5. Reproducibility

Platform to build on?

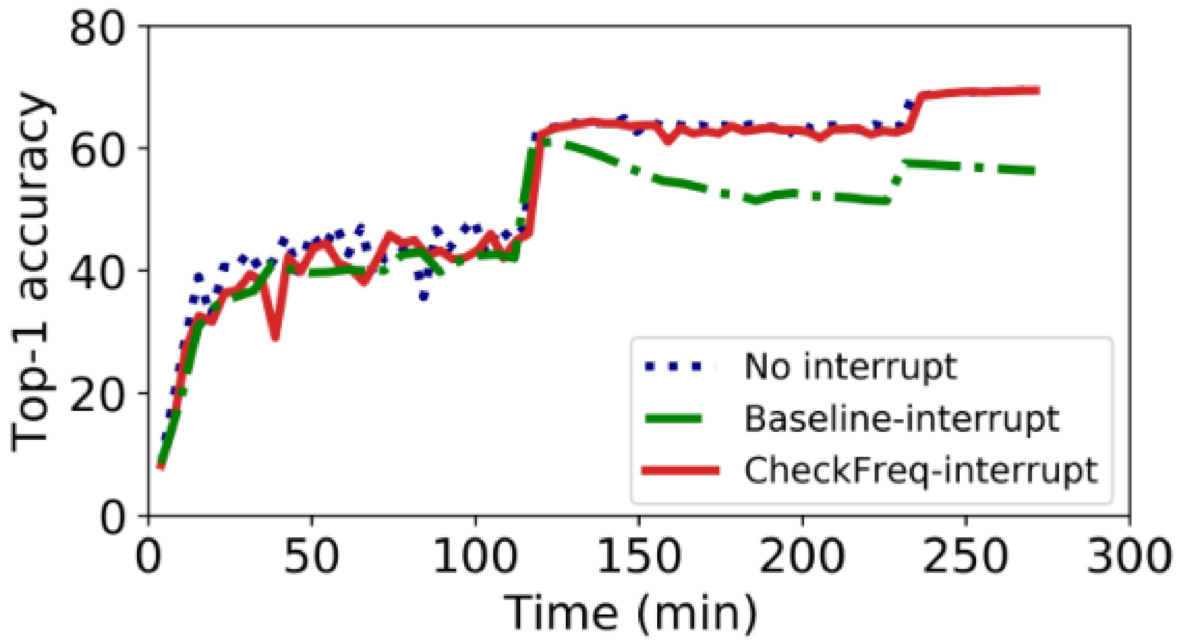
- Know of two disaggregated distributed preprocessing services: tf.data service and Meta's DPP
- Use Cachew (built on tf.data service) as it is open source
- Relieves ML users from the burden of managing compute, memory and storage infrastructure for ML input preprocessing
- Builds on top of tf.data service and provides:
  - Distributed disaggregated multi-tenant input pre-processing (Req1)
  - Supports tf.data pipelines (Req4)
  - Autocaching and autoscaling

Mitigating Worker Failure Impact on Accuracy

How often should a client see an example per epoch?

- No-guarantees, except for the same epoch size
  - See figure on next-slide, accuracy suffers, double-digit drop
- At-most-once
  - Could lead to some examples never being seen
  - Problematic with highly class-imbalanced datasets
- Exactly-once
  - No possibility of model performance degrading due to “invisible” pre-processing failures

CheckFreq Accuracy Drop



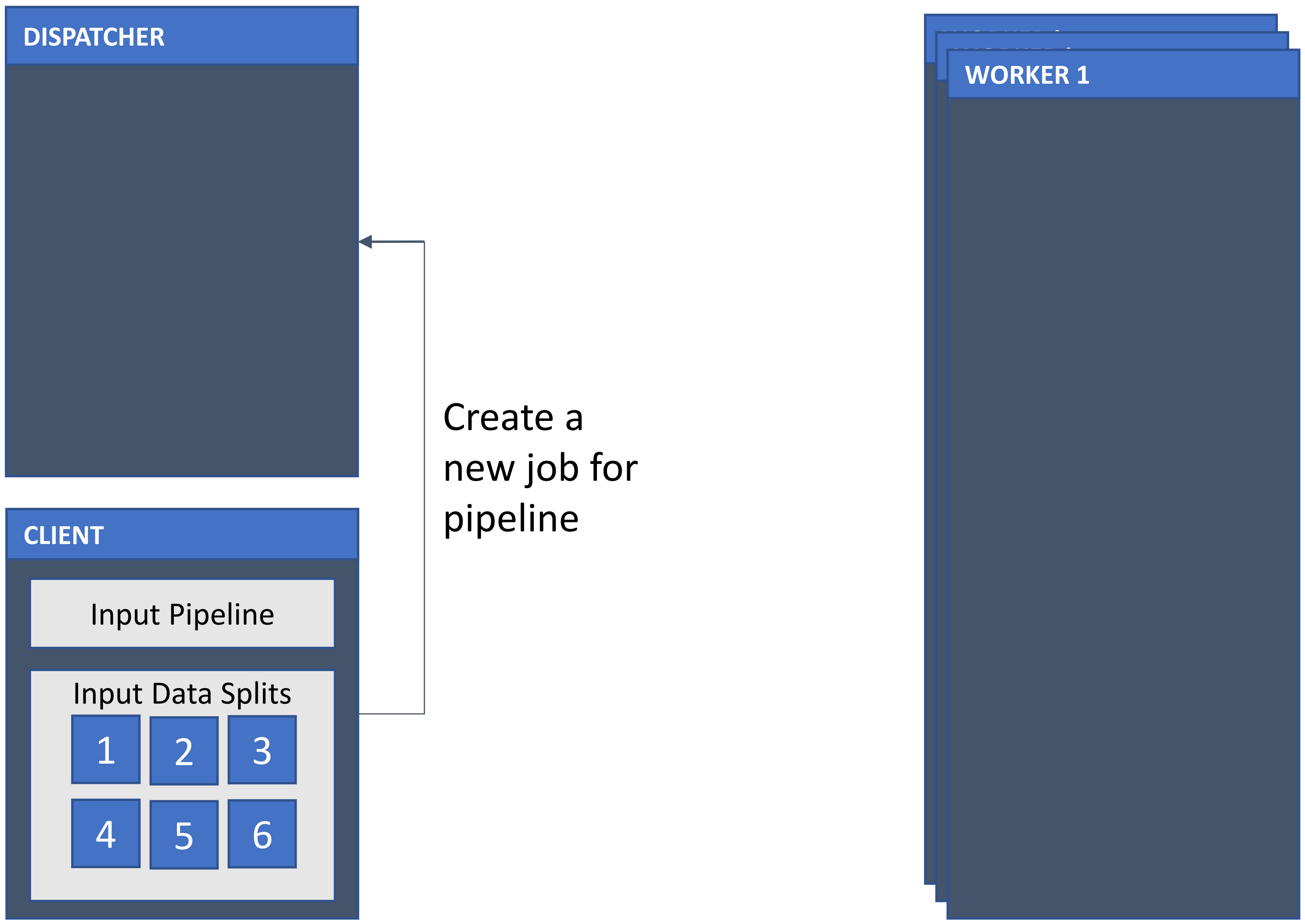
Mohan, J., Phanishayee, A., & Chidambaram, V. (2021, February). CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST 2021).

Takeaway

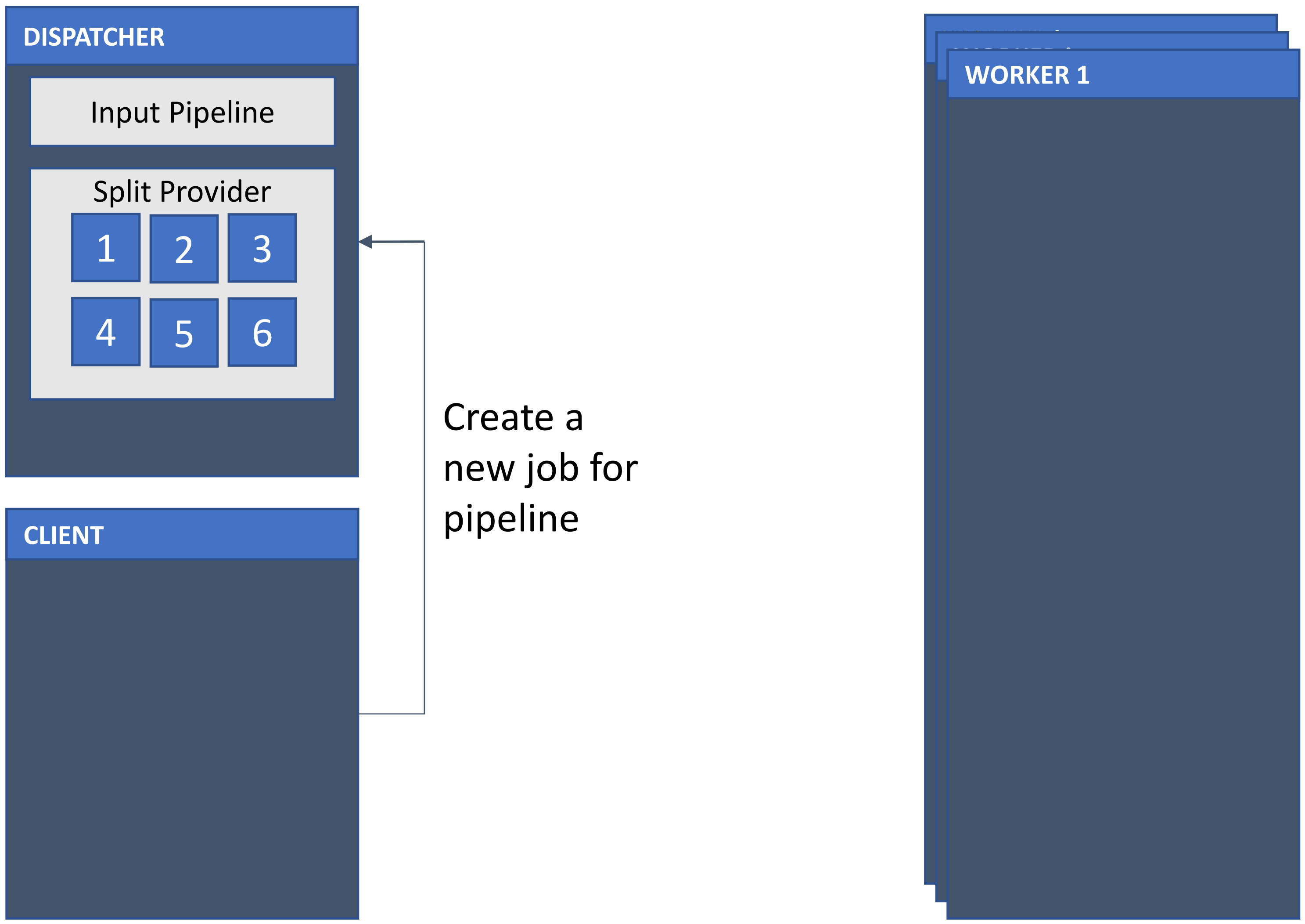
Duplicated / missing examples lead to a 13% drop in accuracy

Cachew Illustrated

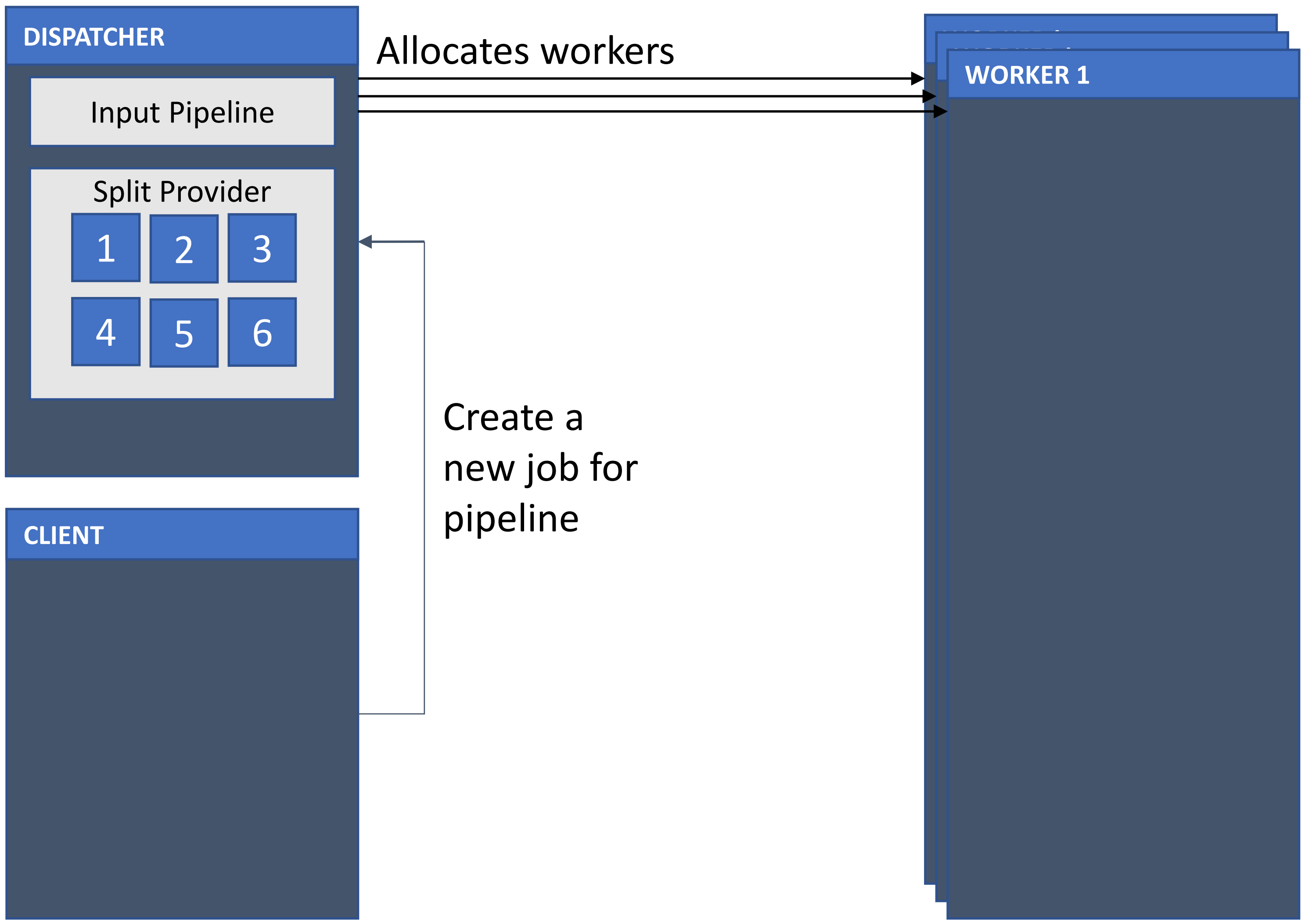
Cachew Illustrated



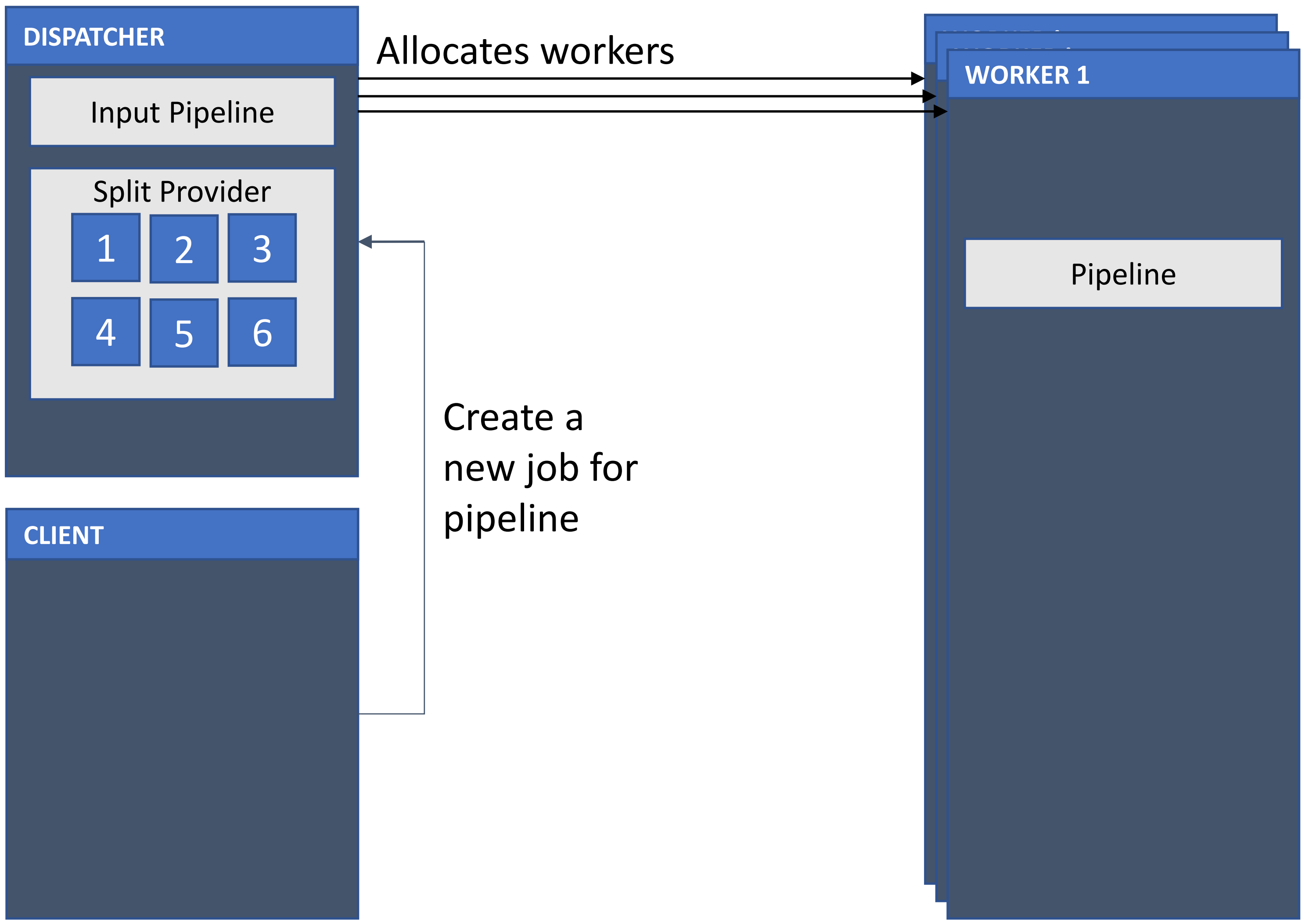
Cachew Illustrated



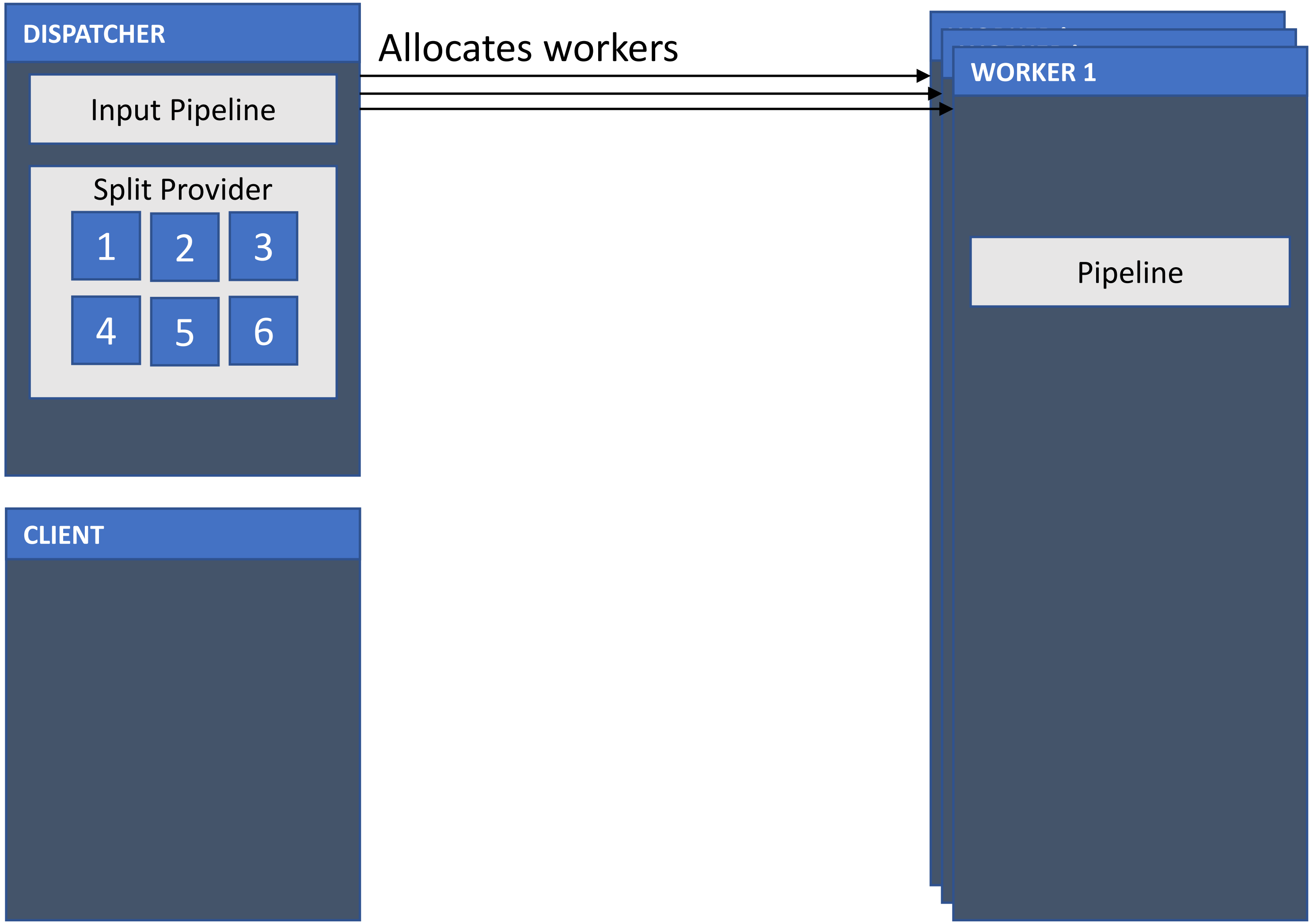
Cachew Illustrated



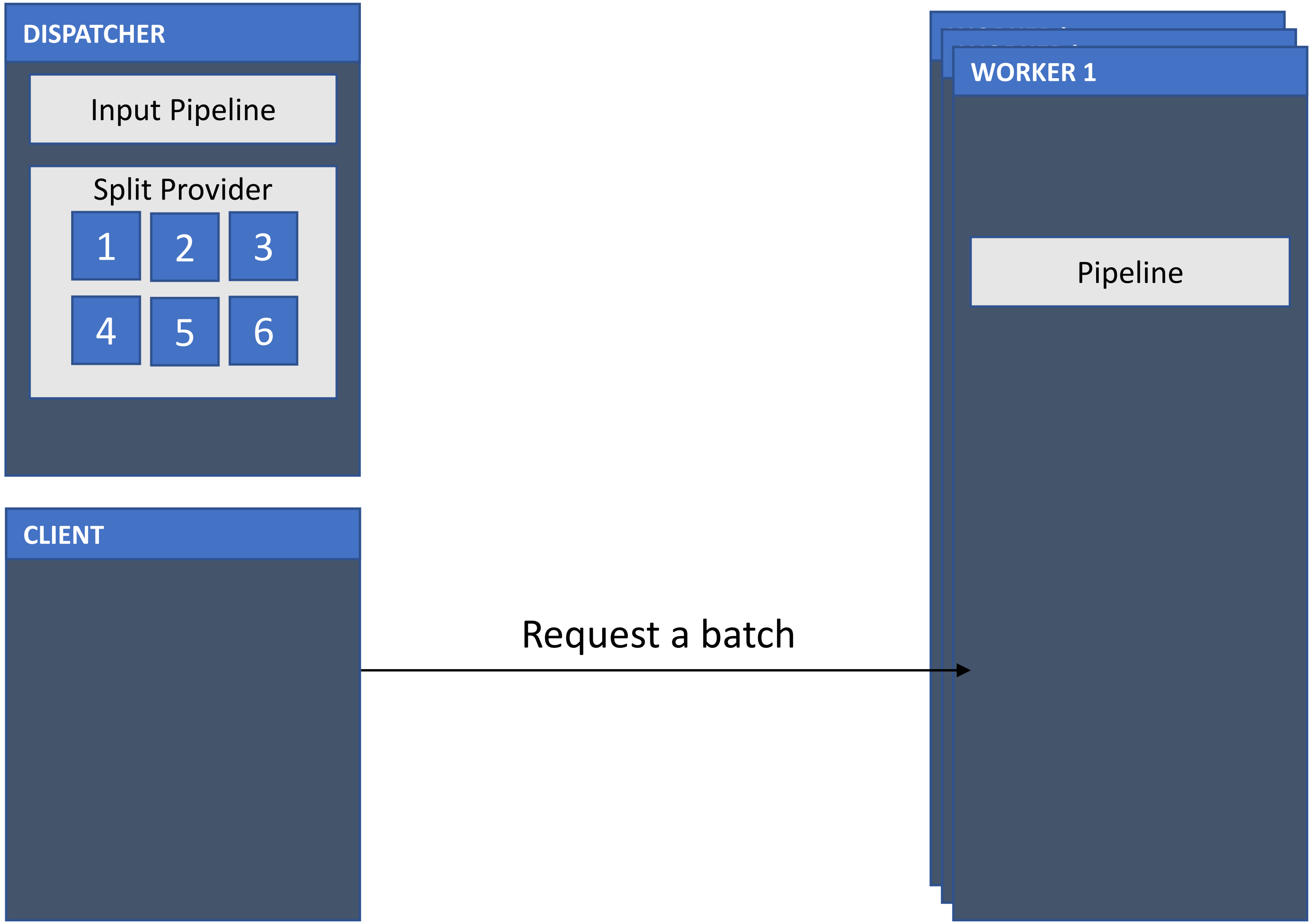
Cachew Illustrated



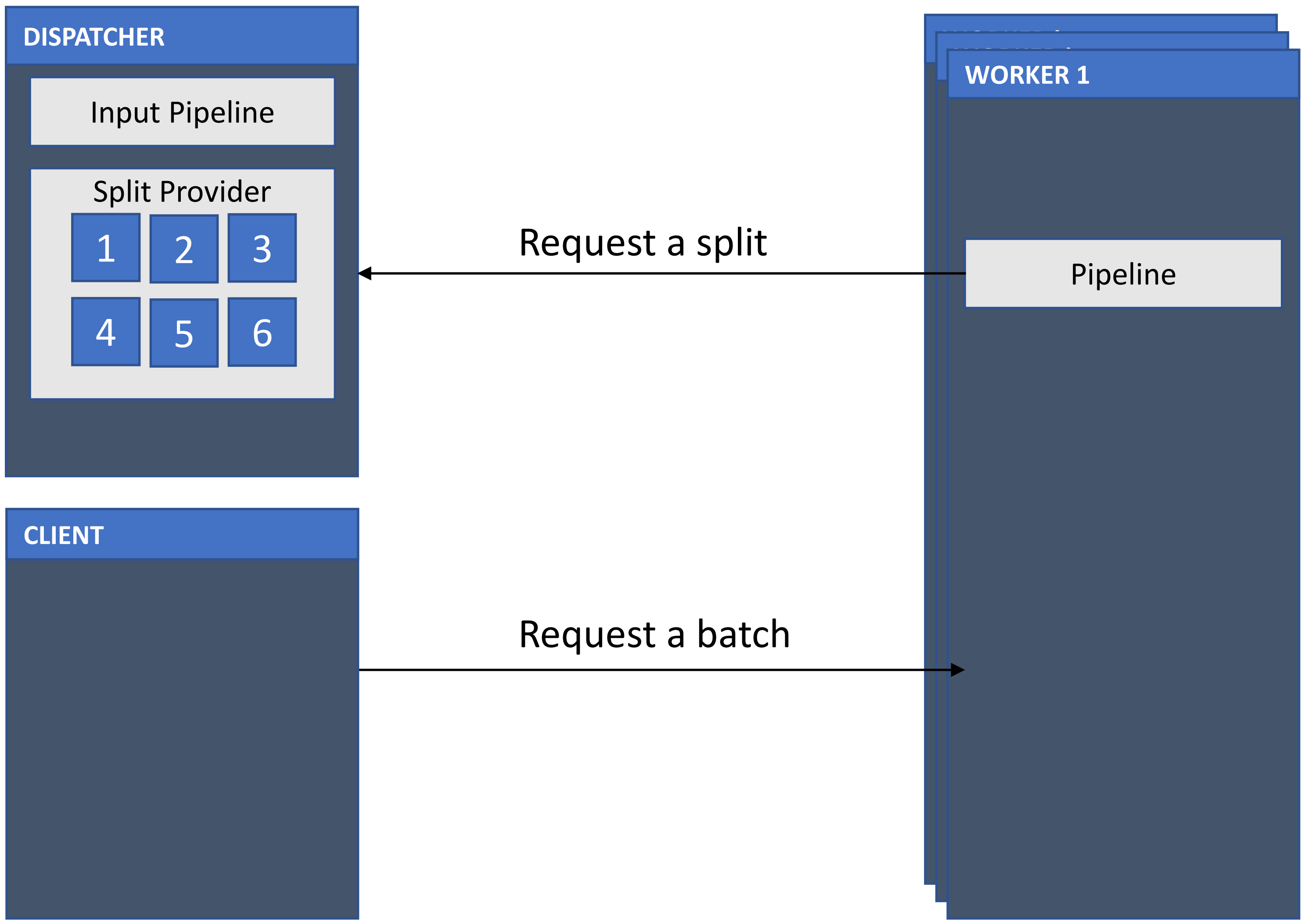
Cachew Illustrated



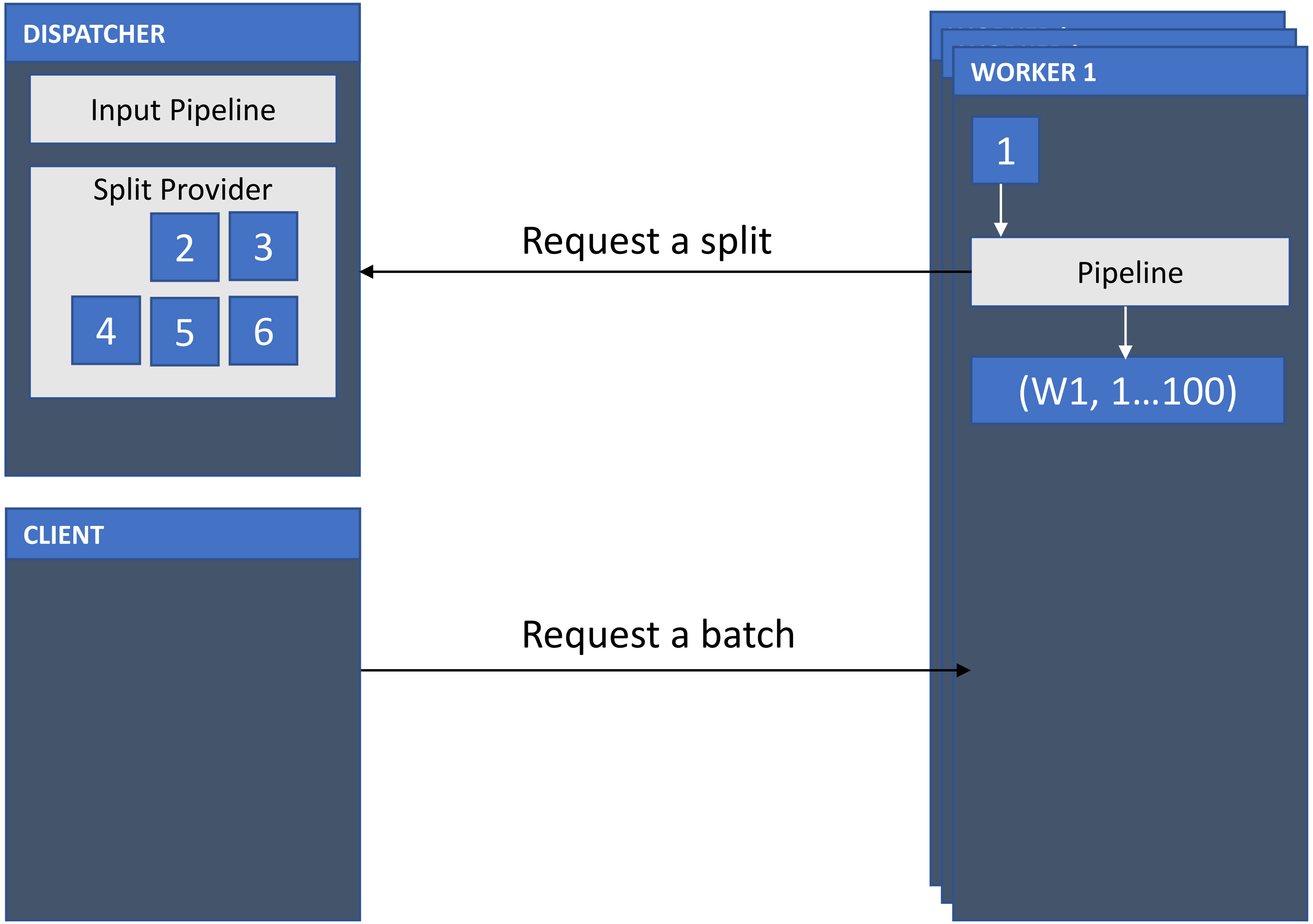
Cachew Illustrated



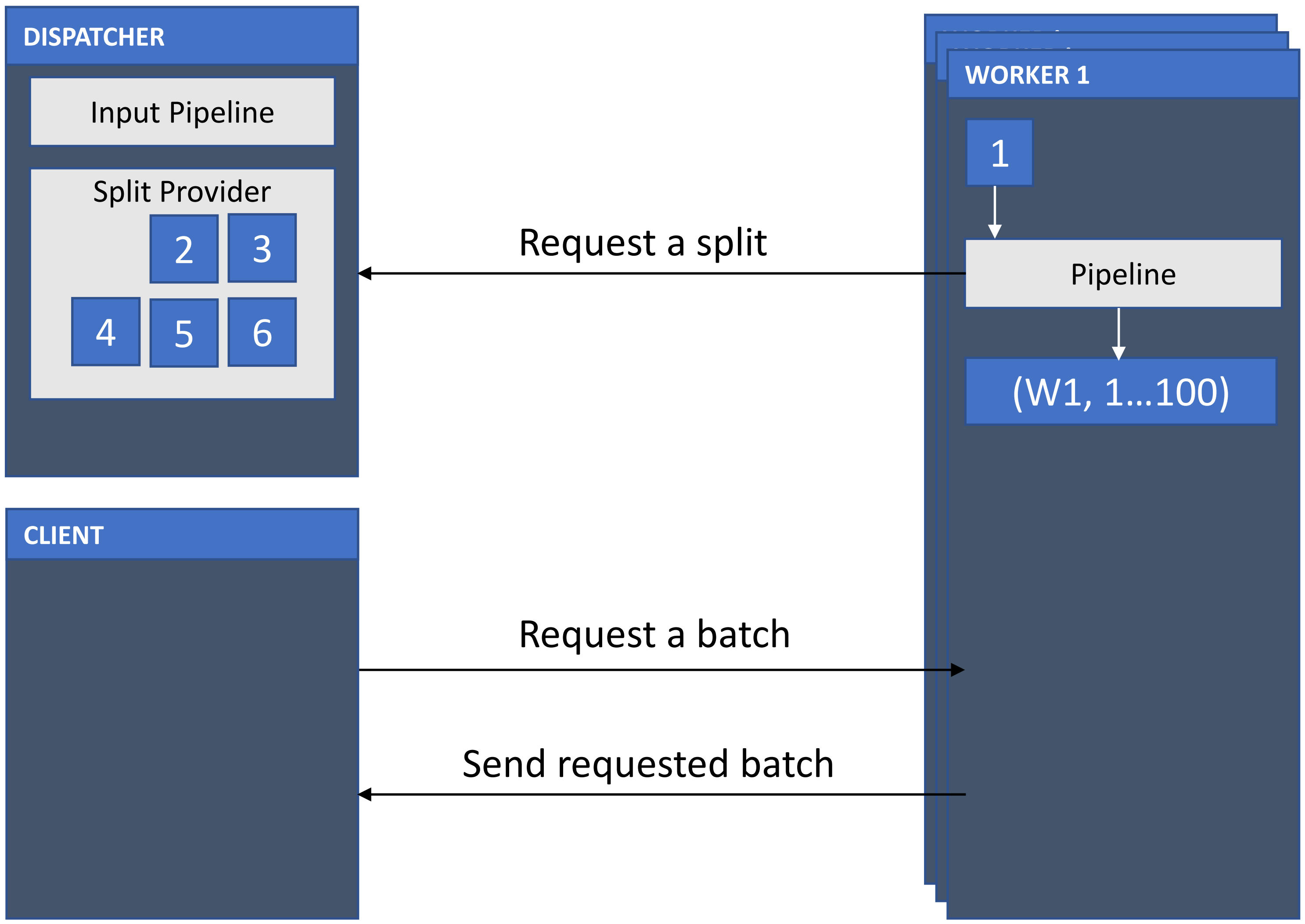
Cachew Illustrated



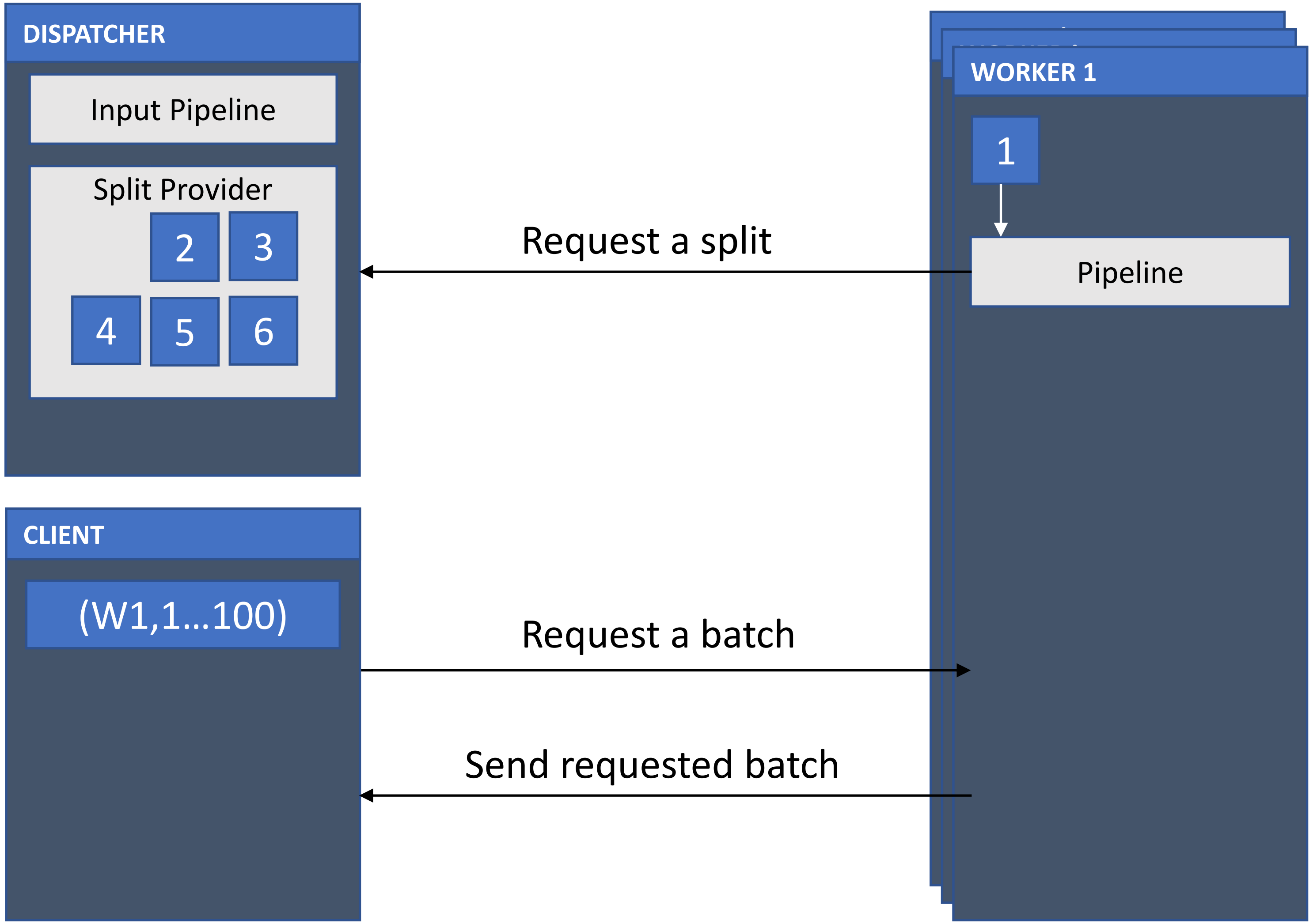
Cachew Illustrated



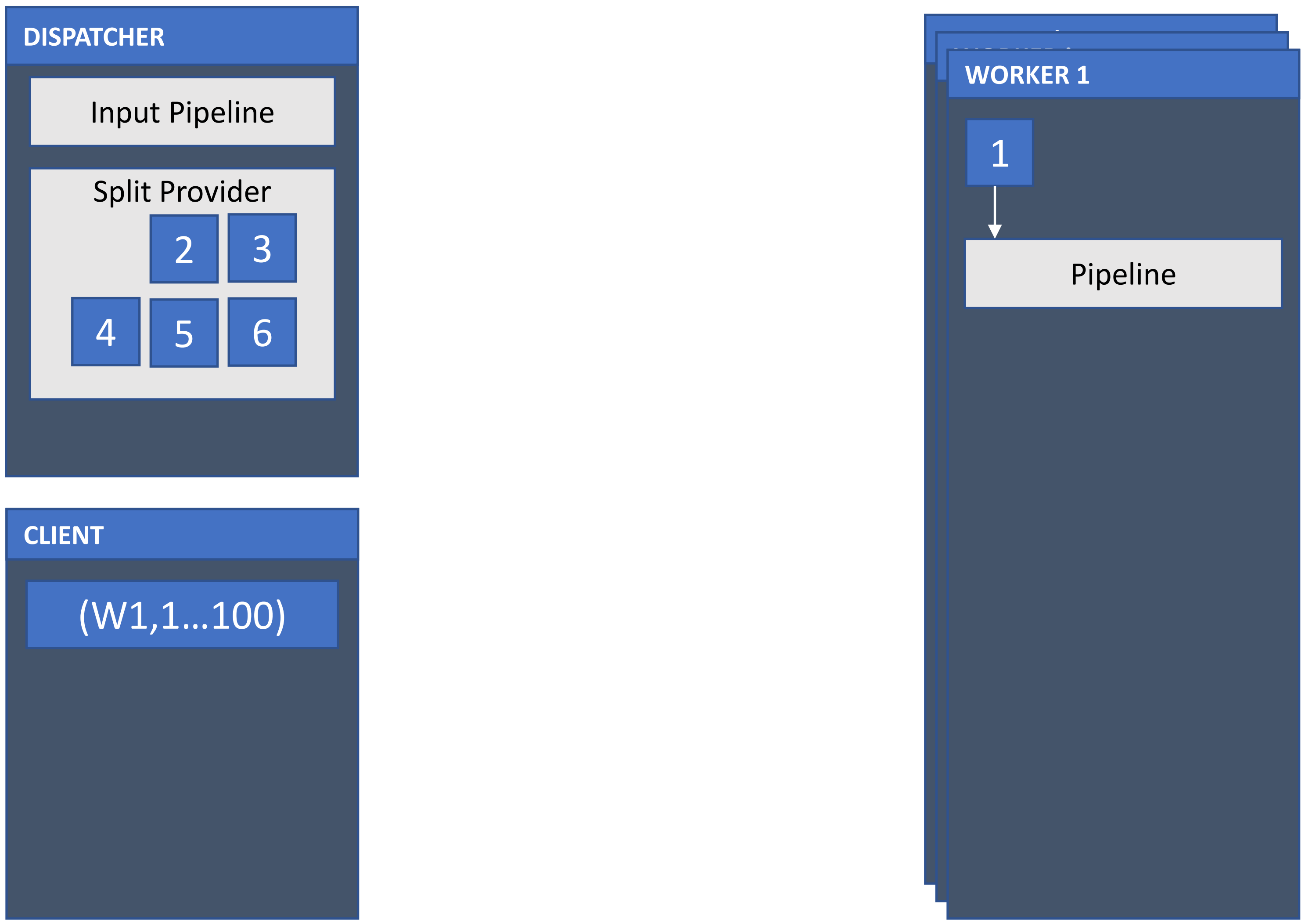
Cachew Illustrated



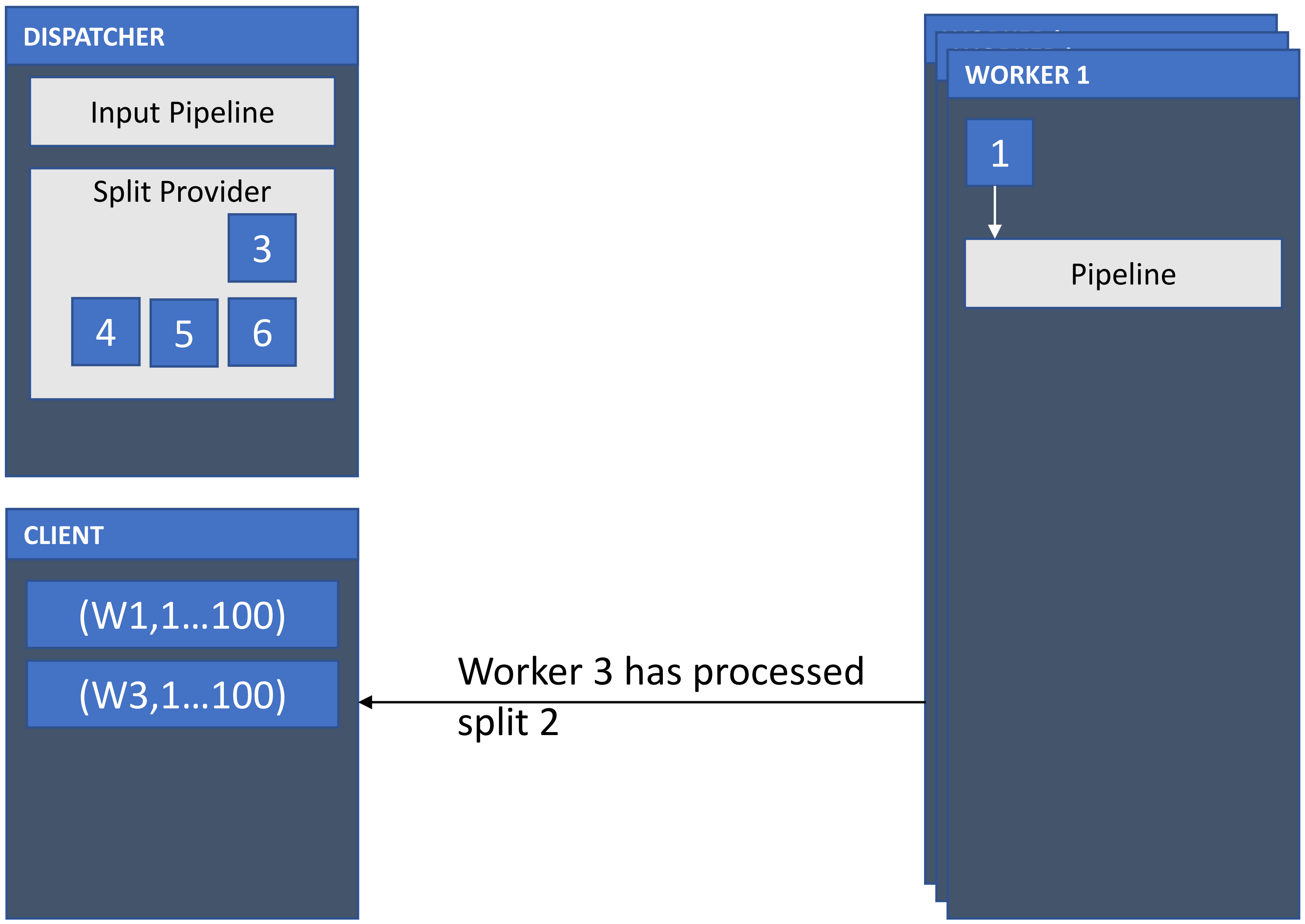
Cachew Illustrated



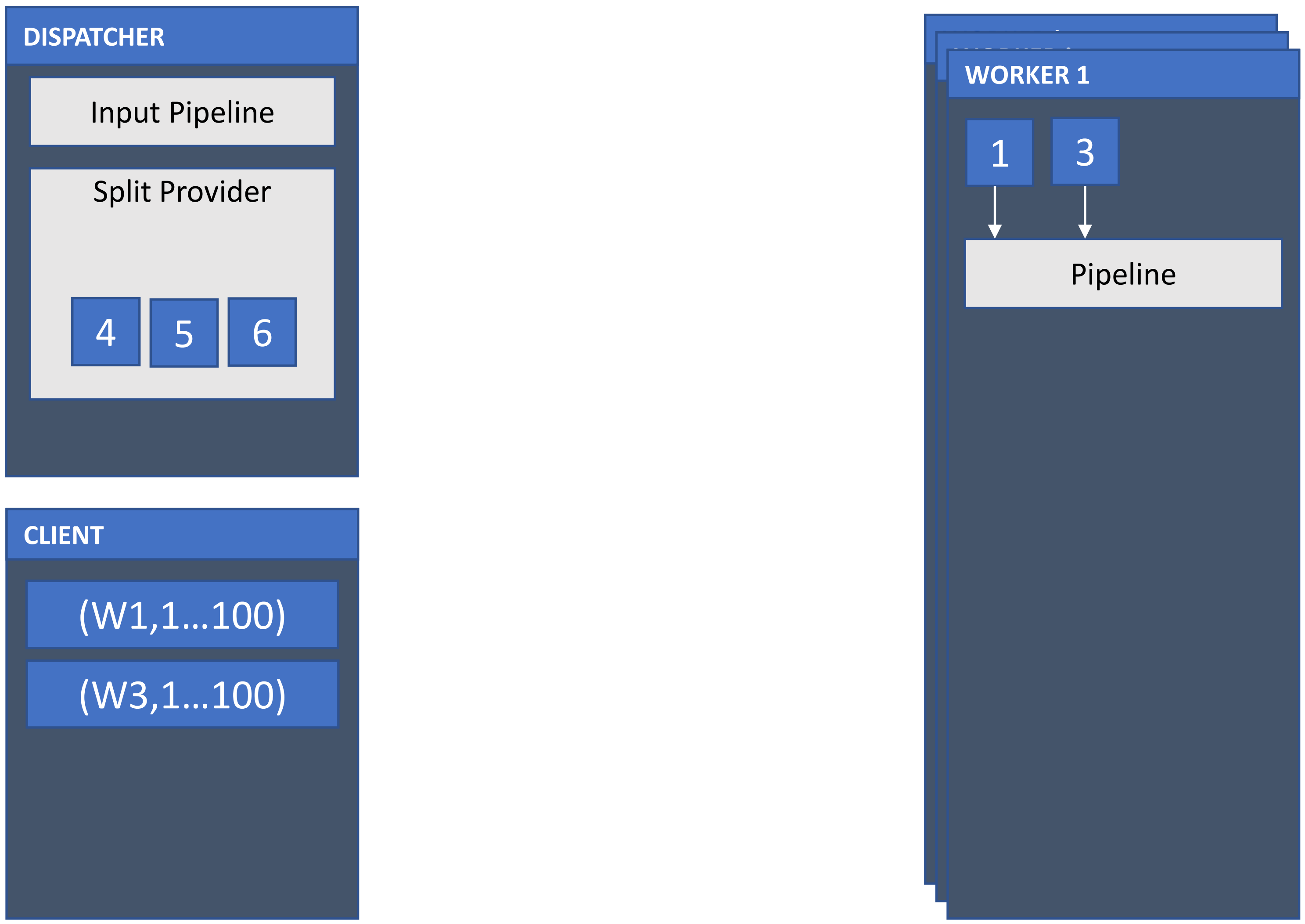
Cachew Illustrated



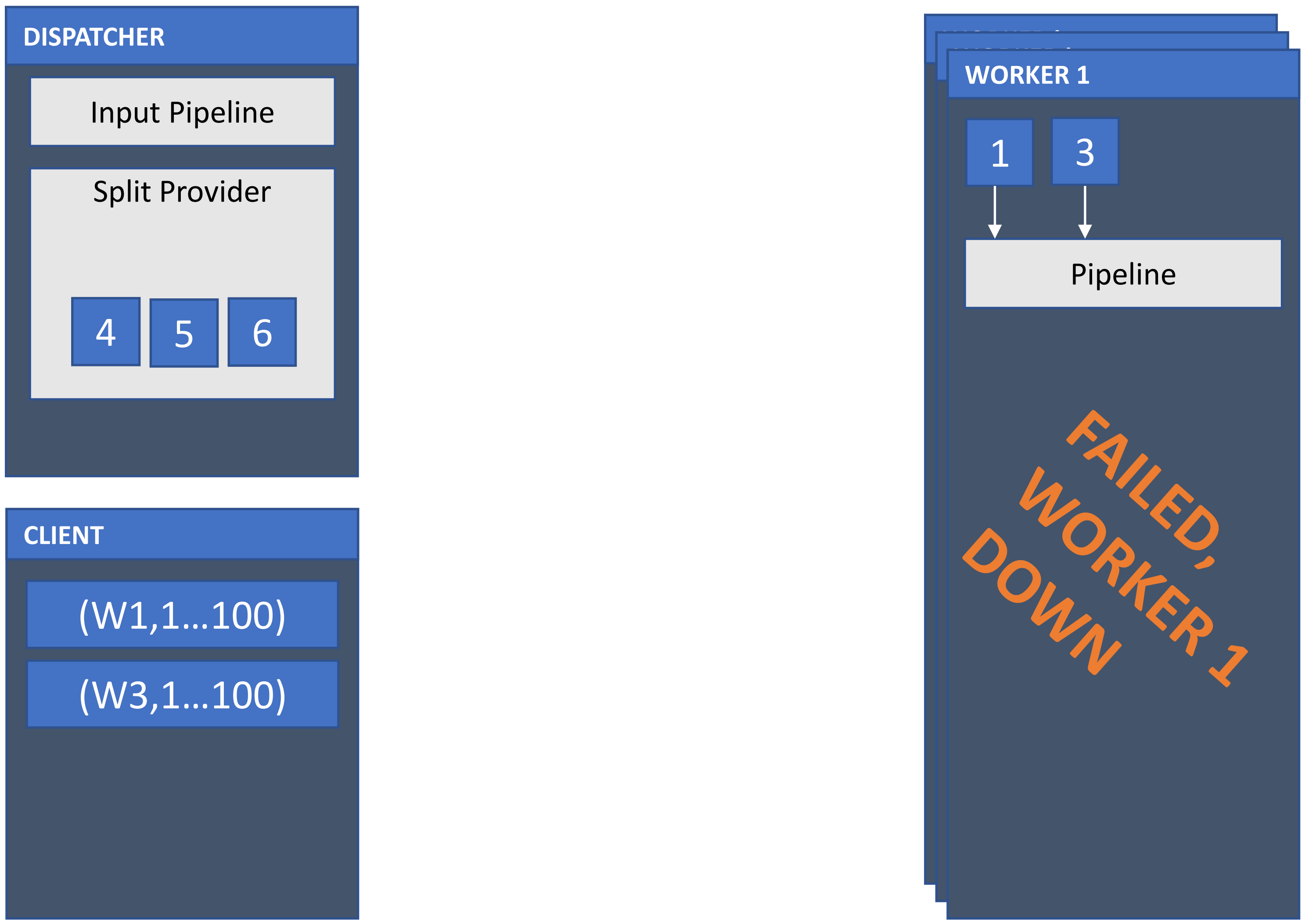
Cachew Illustrated



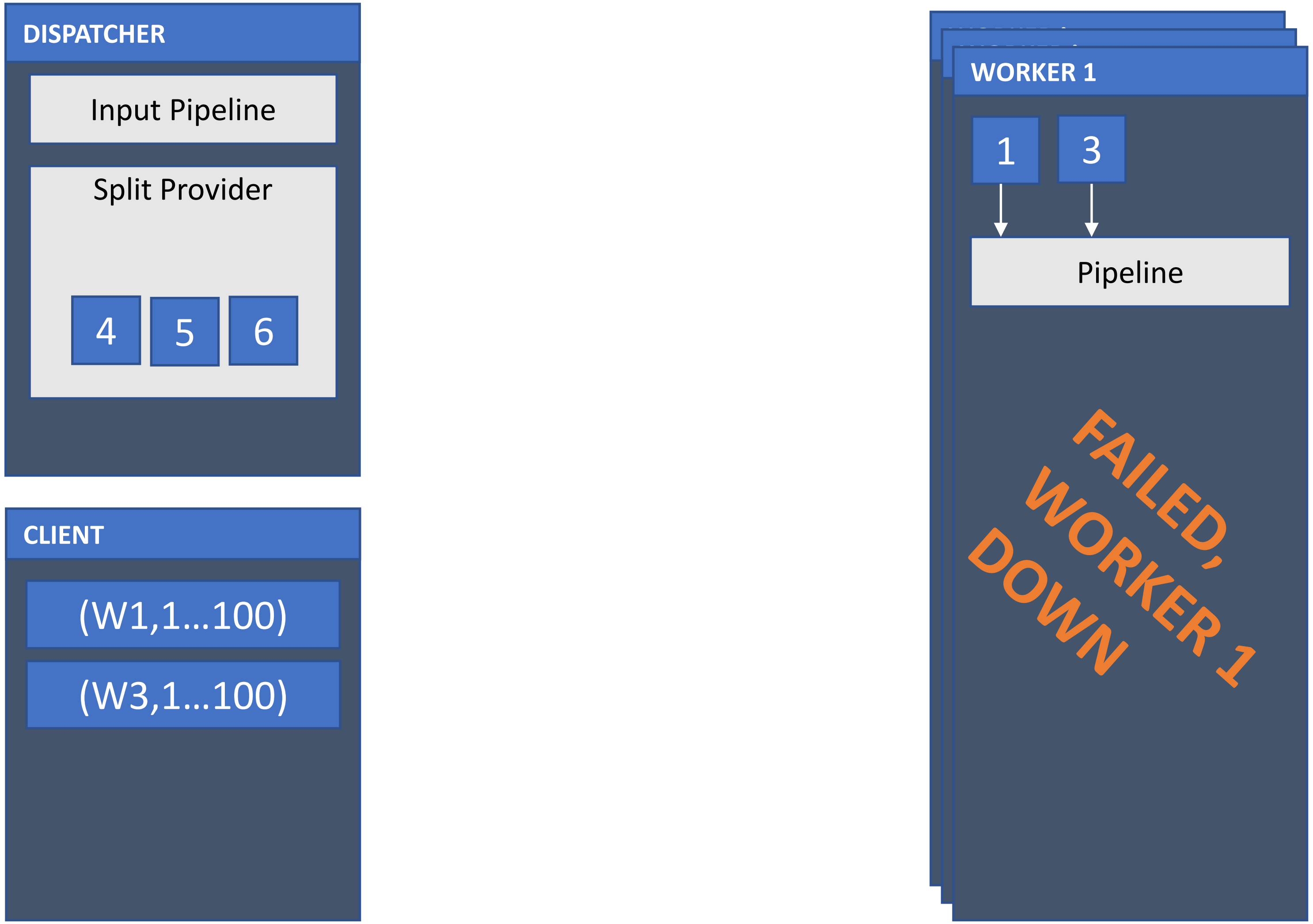
Cachew Illustrated



Cachew Illustrated



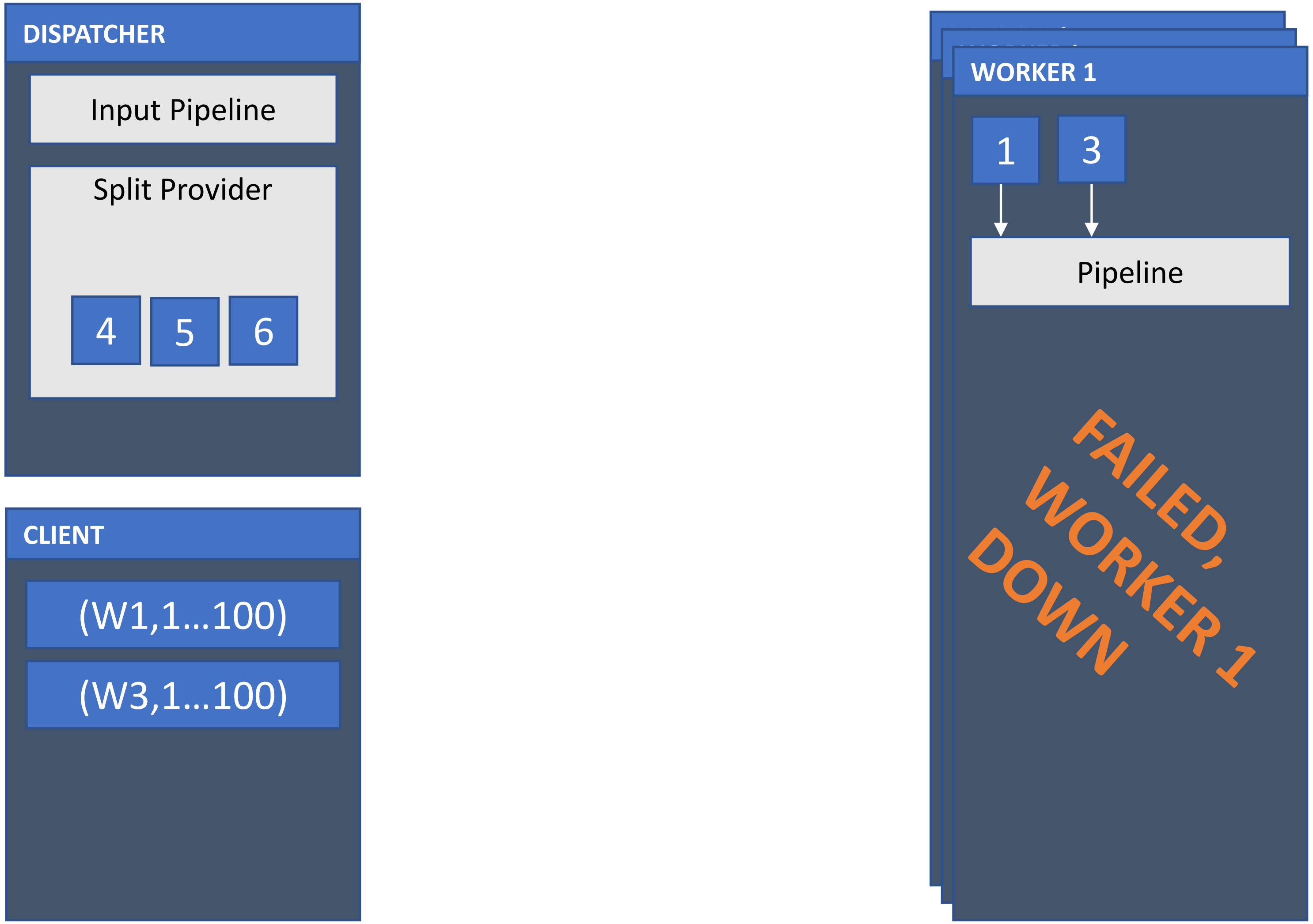
Cachew Illustrated



How can we react?

- Skip 3: at-most-once
- Recompute 1 & 3, transmit everything: at-least-once
- Recompute 1 & 3, but skip 1 at the client: exactly-once

Cachew Illustrated



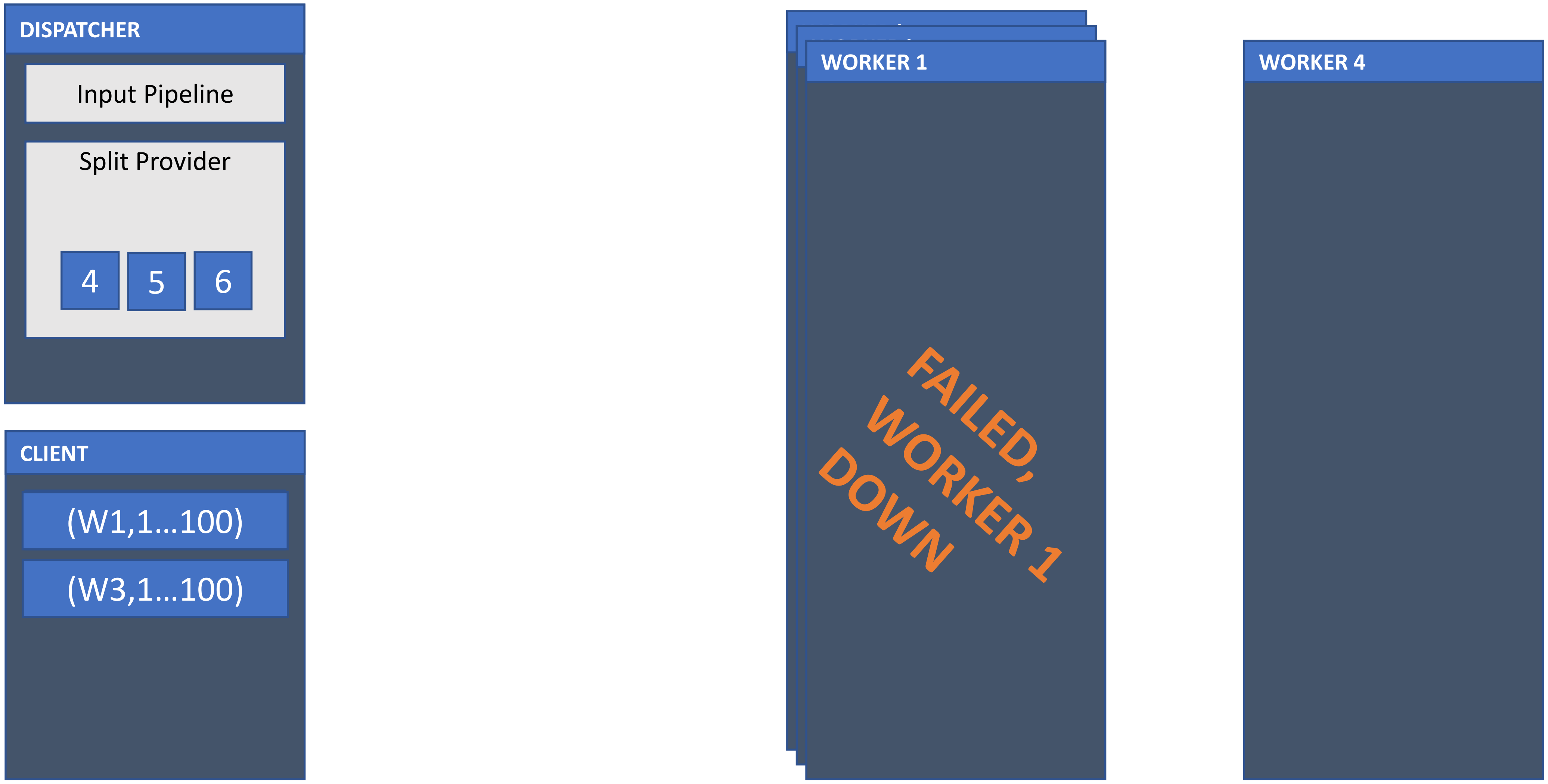
How can we react?

- Skip 3: at-most-once
- Recompute 1 & 3, transmit everything: at-least-once
- Recompute 1 & 3, but skip 1 at the client: exactly-once

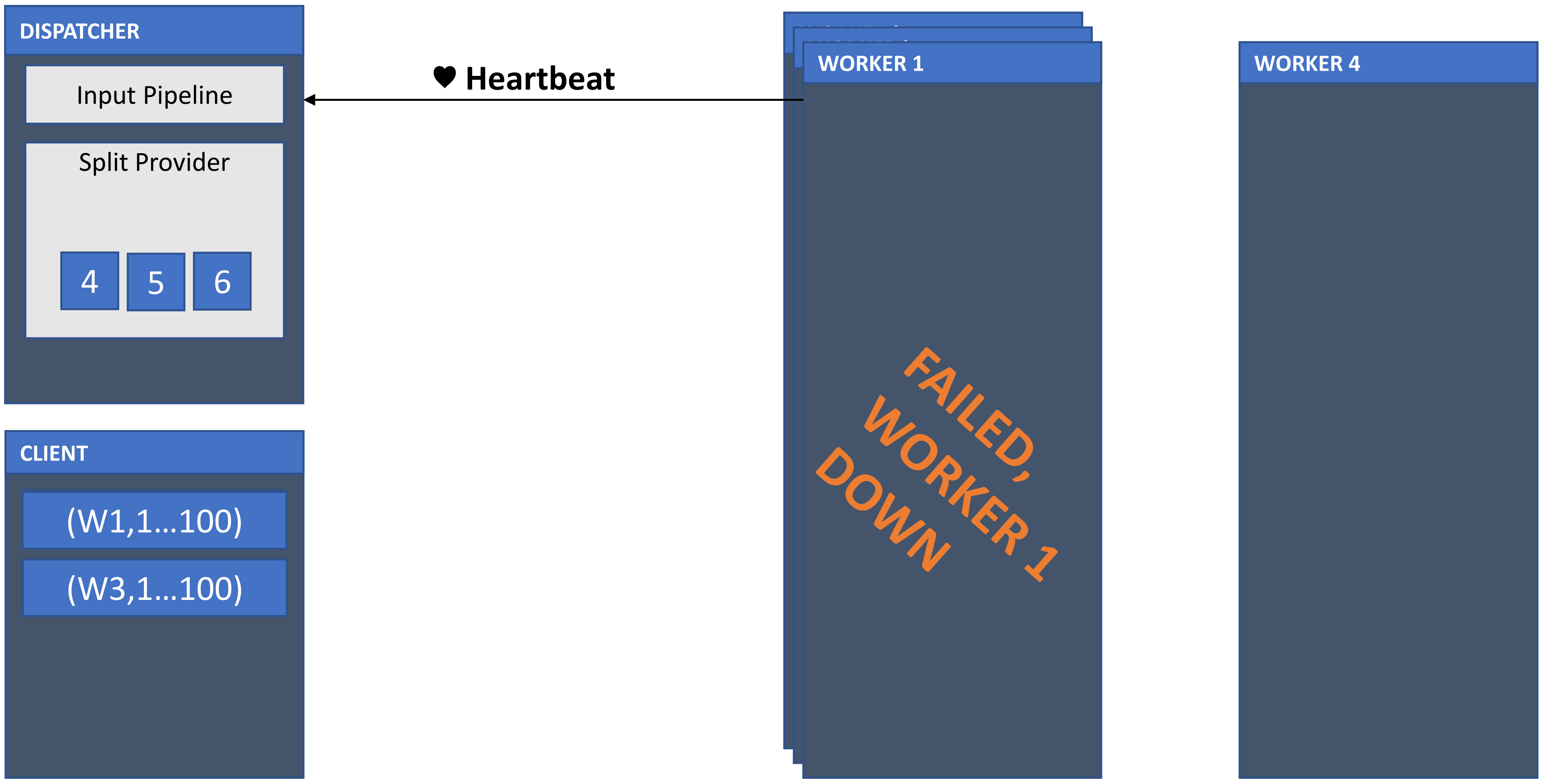
What do we need?

- Dispatcher: Remember split <-> worker association
- Client: Remember expected Batch ID
- Detect failure, failover task to a different worker
- Deterministic ordering

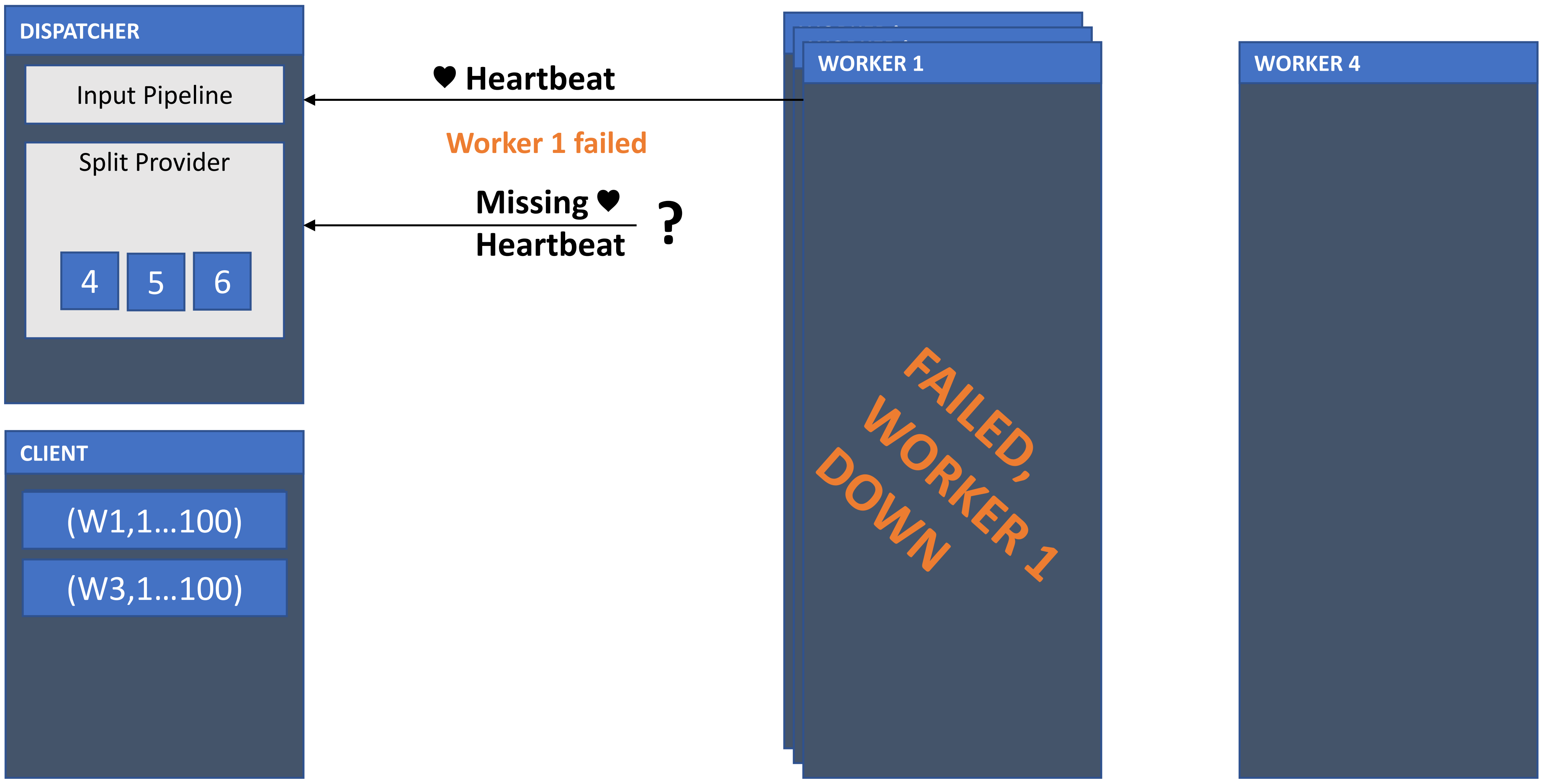
Failover Illustrated



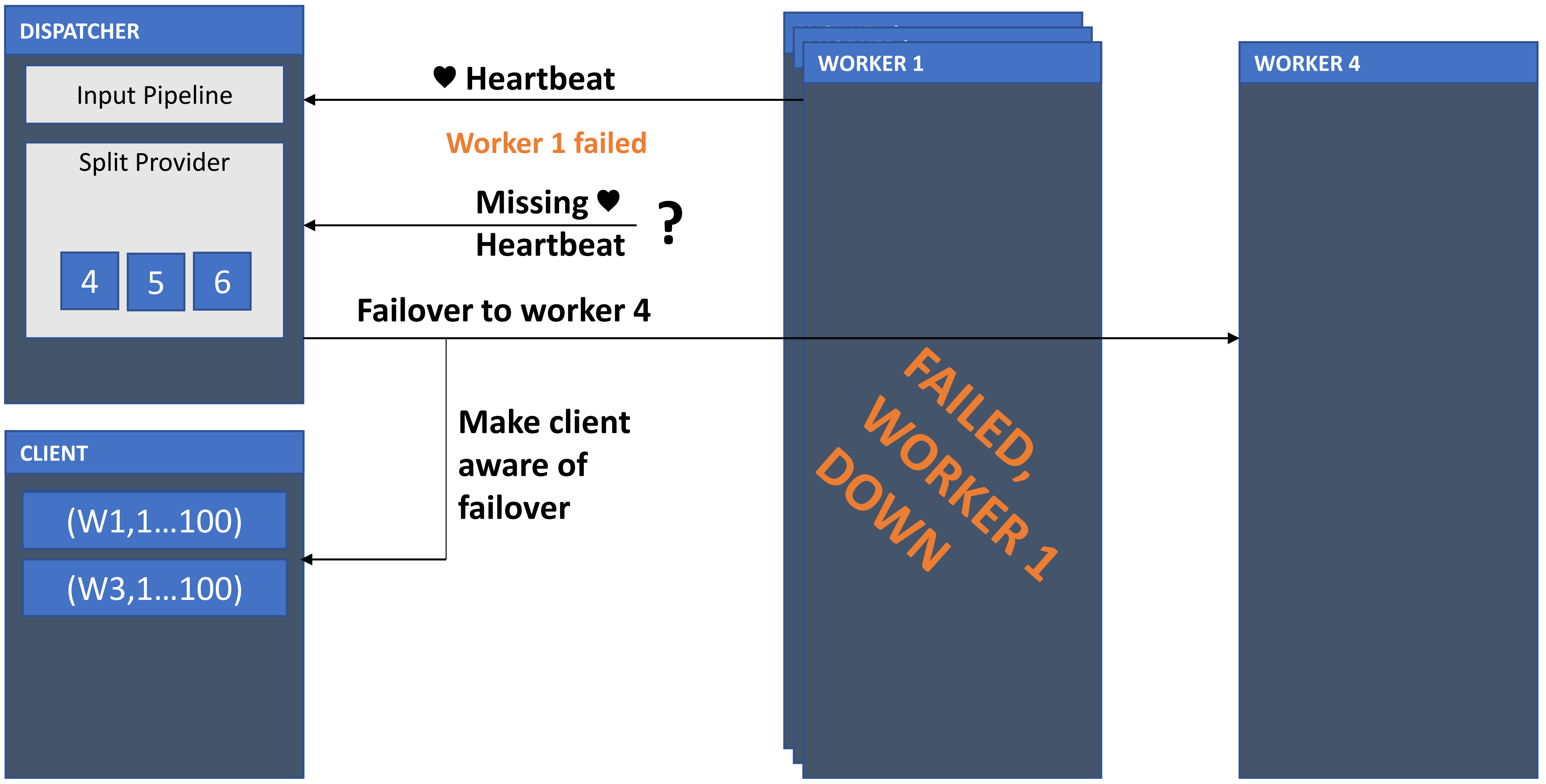
Failover Illustrated



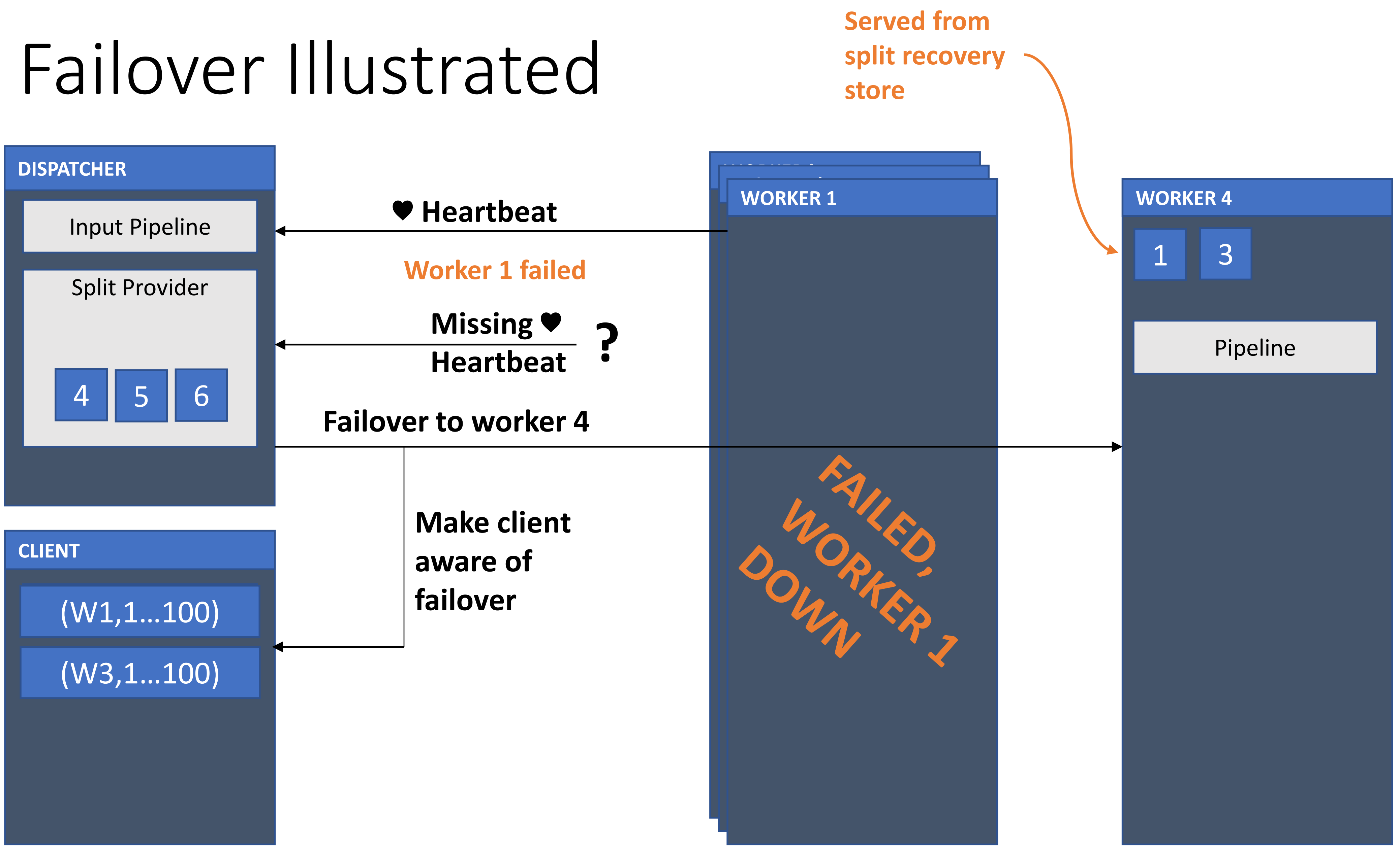
Failover Illustrated



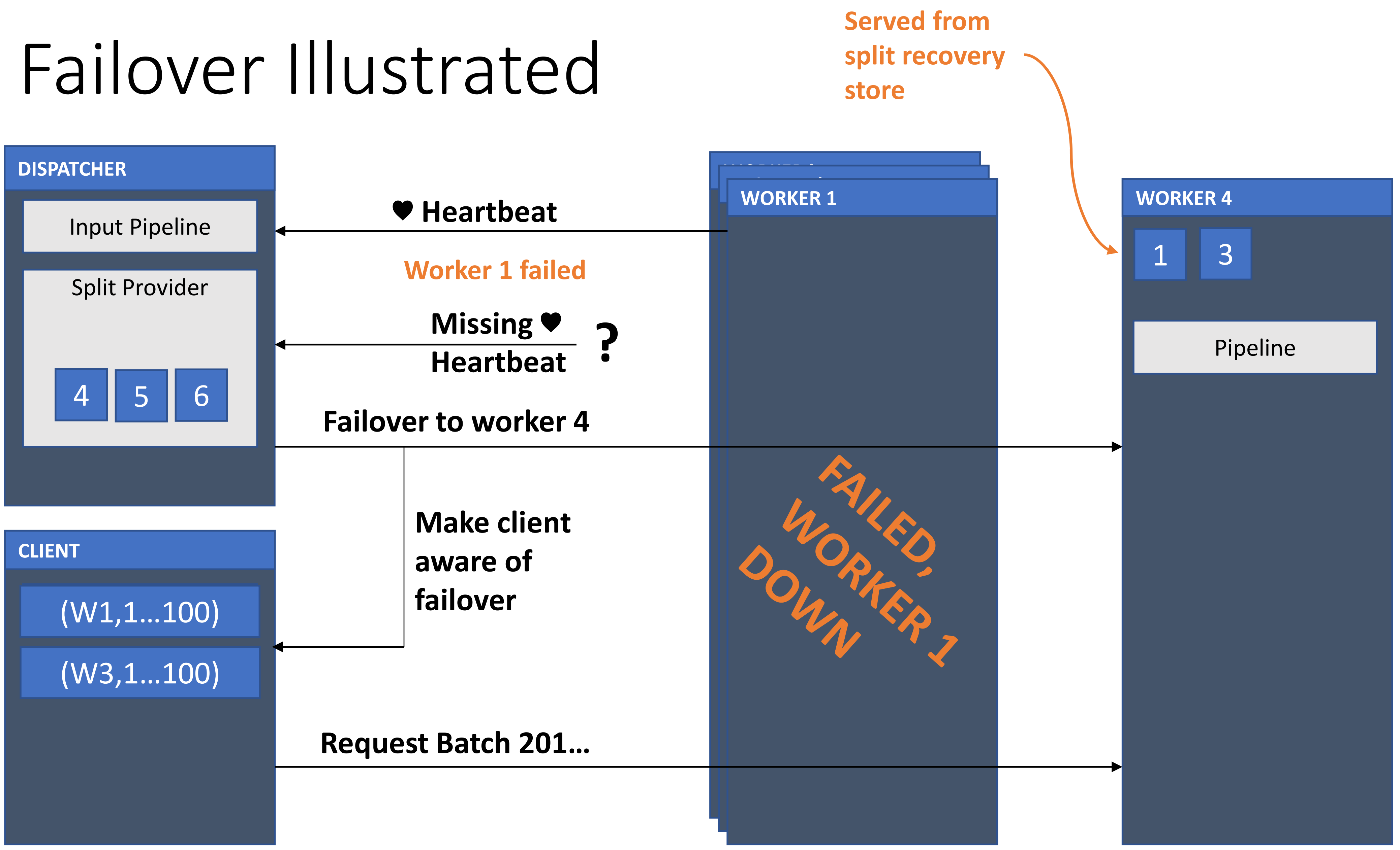
Failover Illustrated



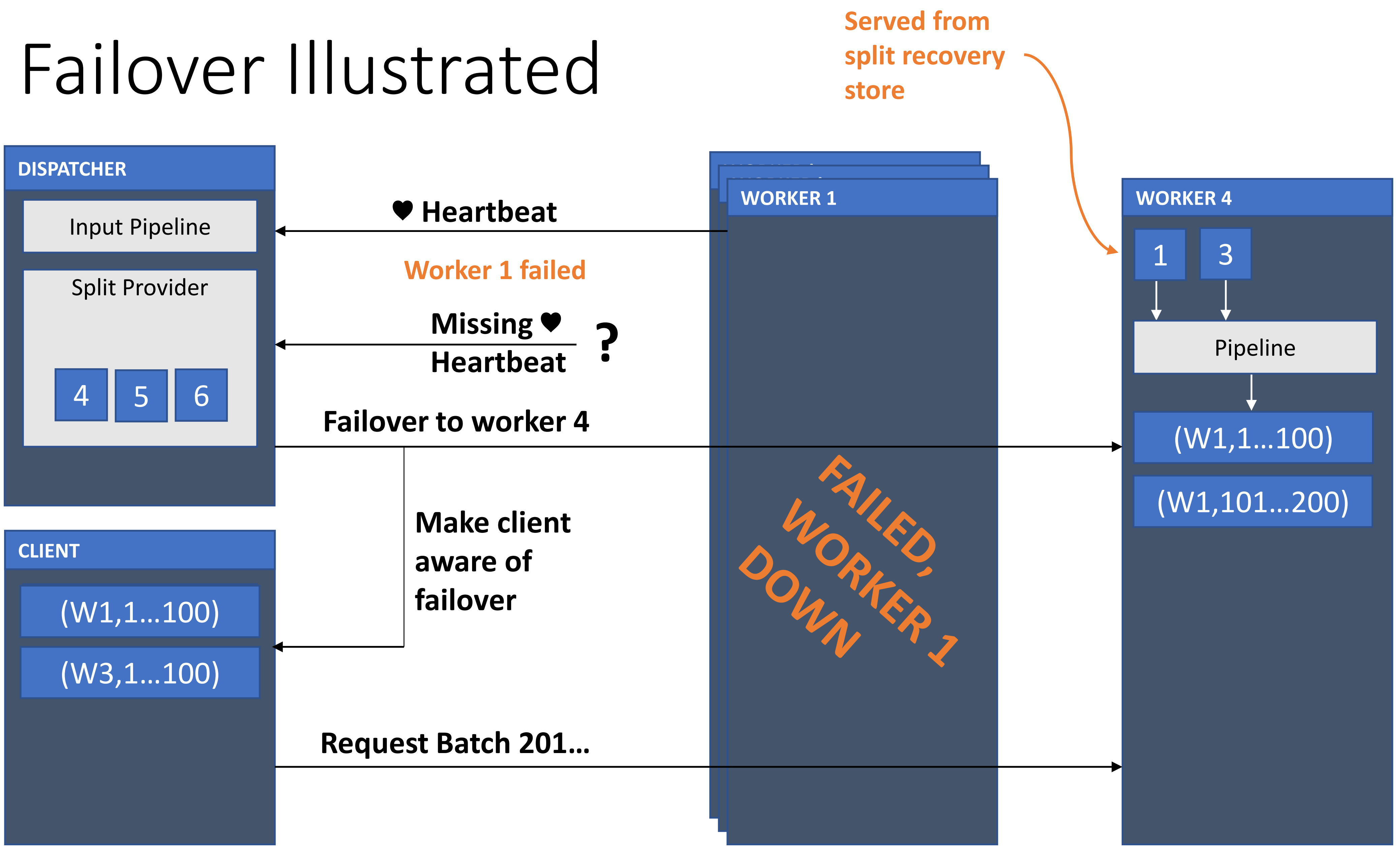
Failover Illustrated



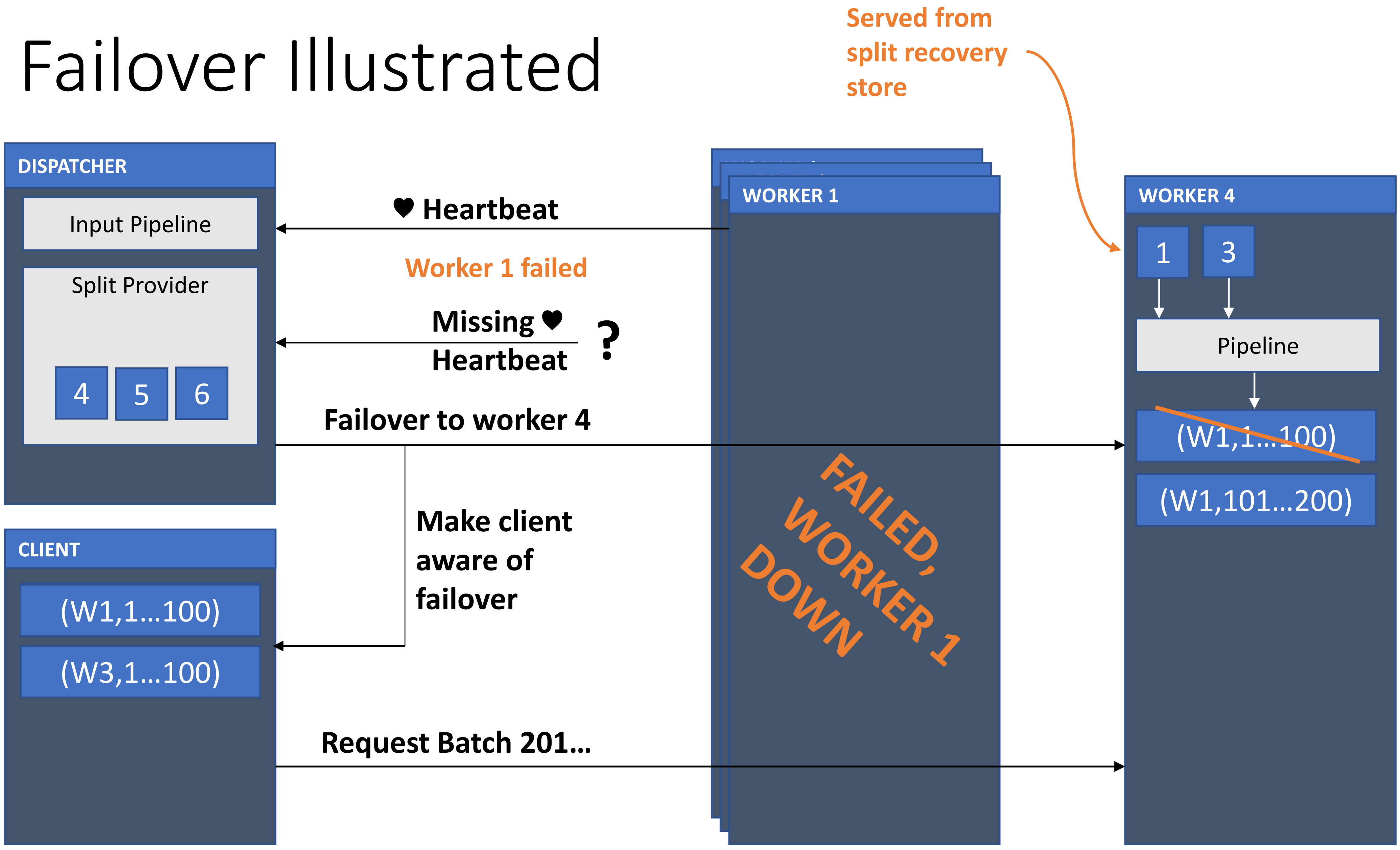
Failover Illustrated



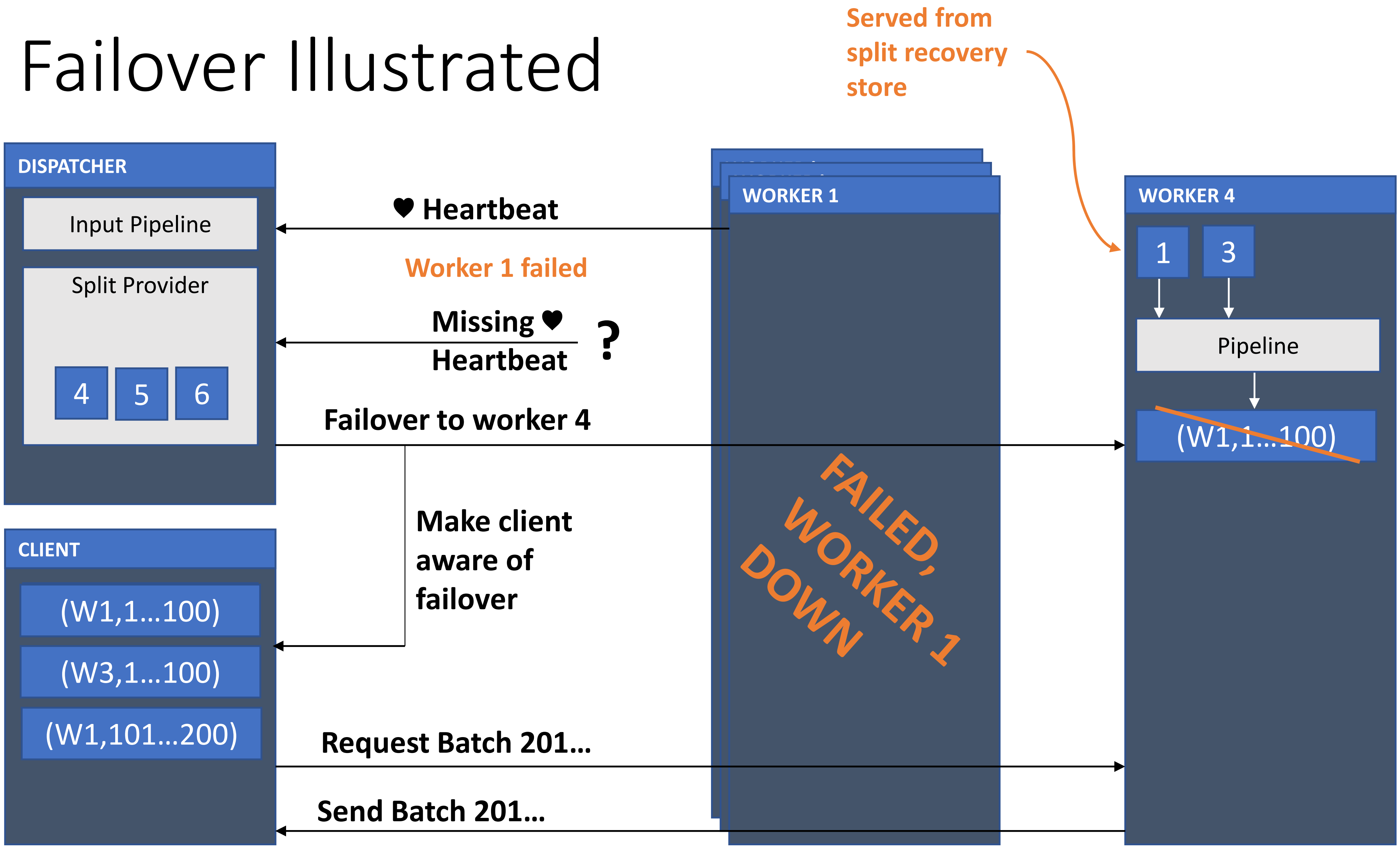
Failover Illustrated



Failover Illustrated



Failover Illustrated



Reducing Overhead

overhead = detection + failover + recomputation

Detection

Fine-tune heartbeat detection mechanisms

~ 1-10s

Failover

Have nodes in hot-standby

Implement “express” messages (circumvent heartbeat-based protocols)

~ 1-10s

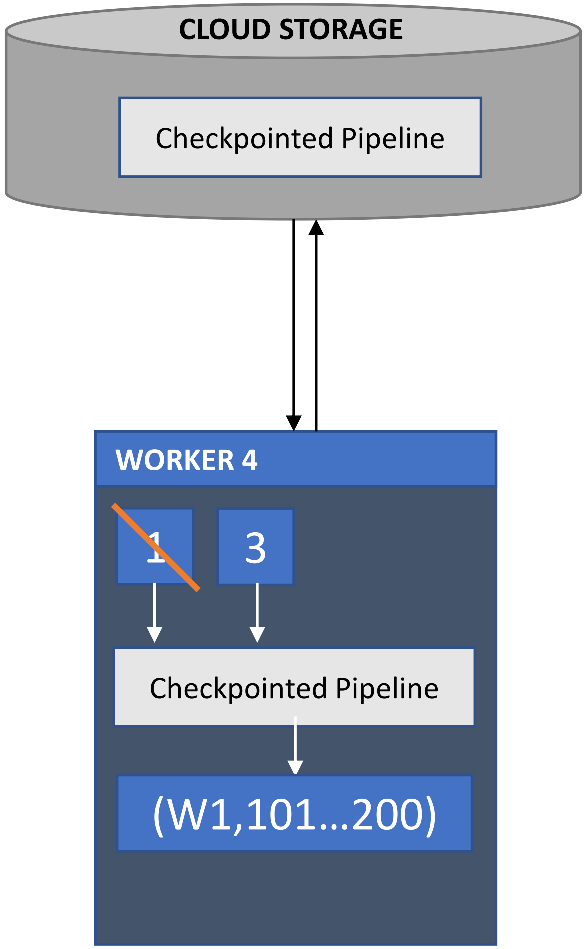
Recomputation

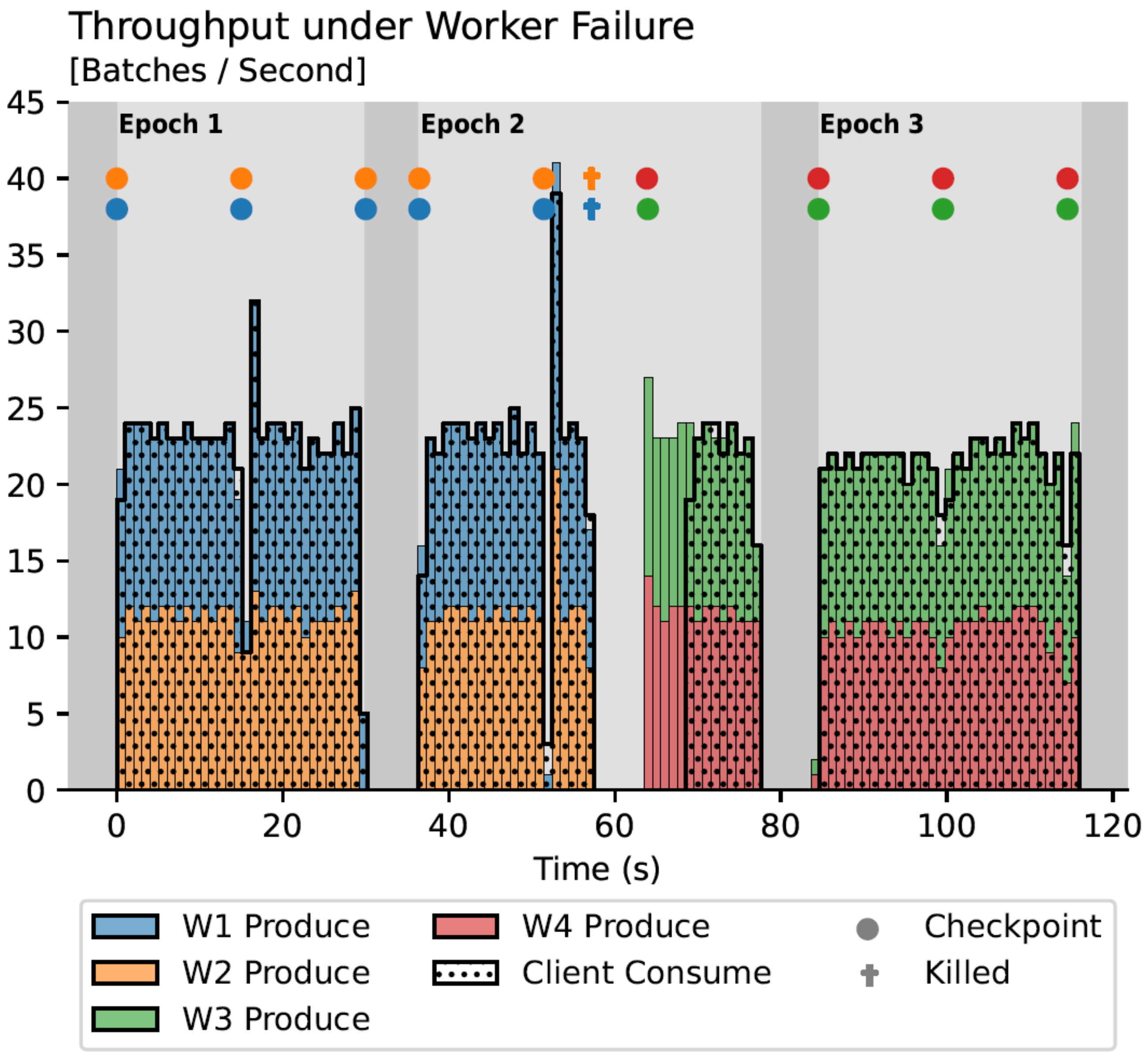
Checkpoint worker state to recompute less

~ ½ epoch

Checkpointing

- Workers regularly checkpoint their pipeline state to GlusterFS (cloud storage)
- This reduces the overhead because we need not recompute everything from scratch (Req3)
- The pipeline needs to be locked to arrive at a consistent state (checkpoint stall)
- Implementation builds on top of existing tf.data checkpointing mechanism

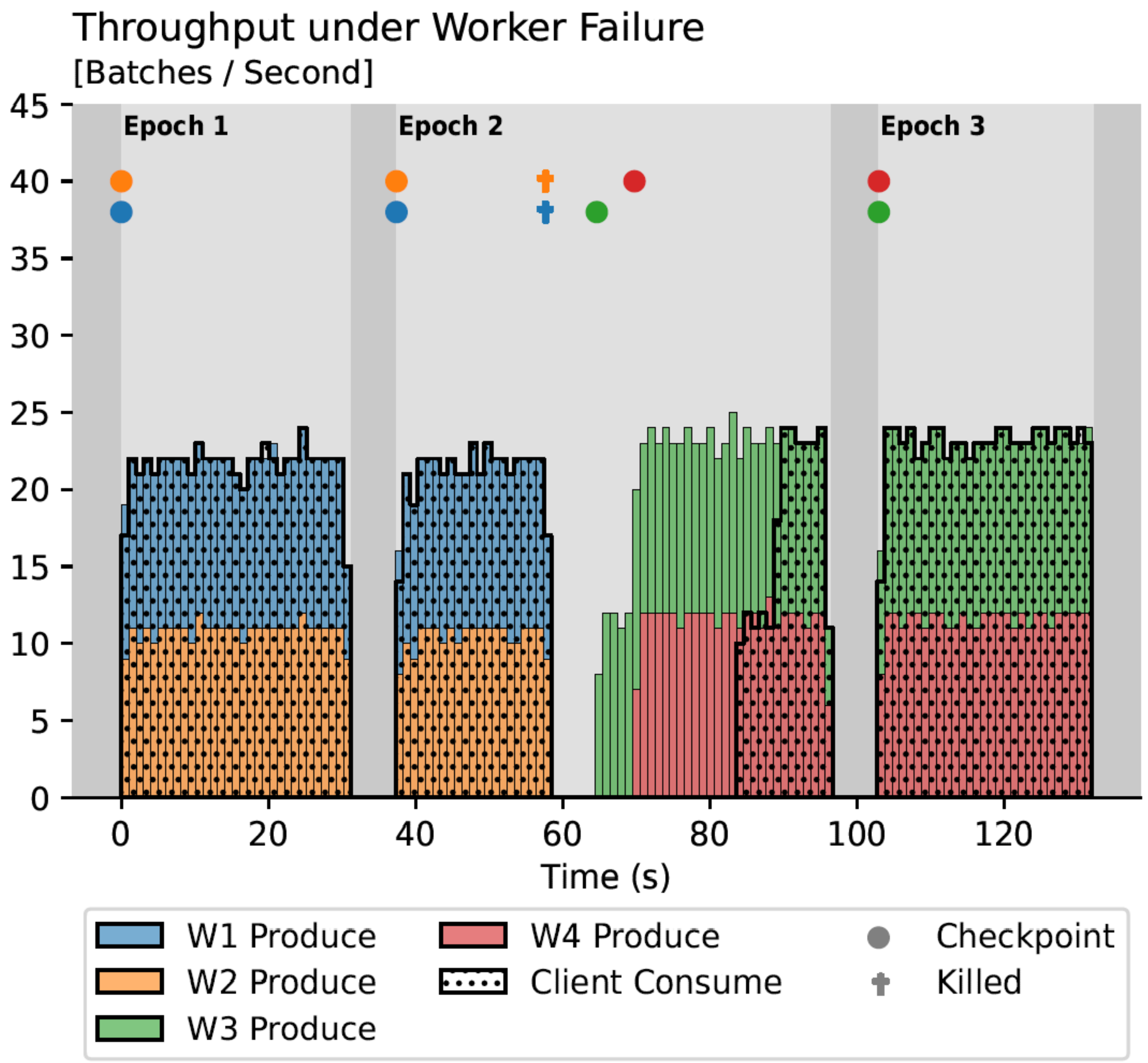




Takeaways

1) Throughput is constant; important characteristic used for scaling/profiling

2) Checkpointing introduced overhead



Takeaways

1) Without any worker checkpointing we have more recomputation

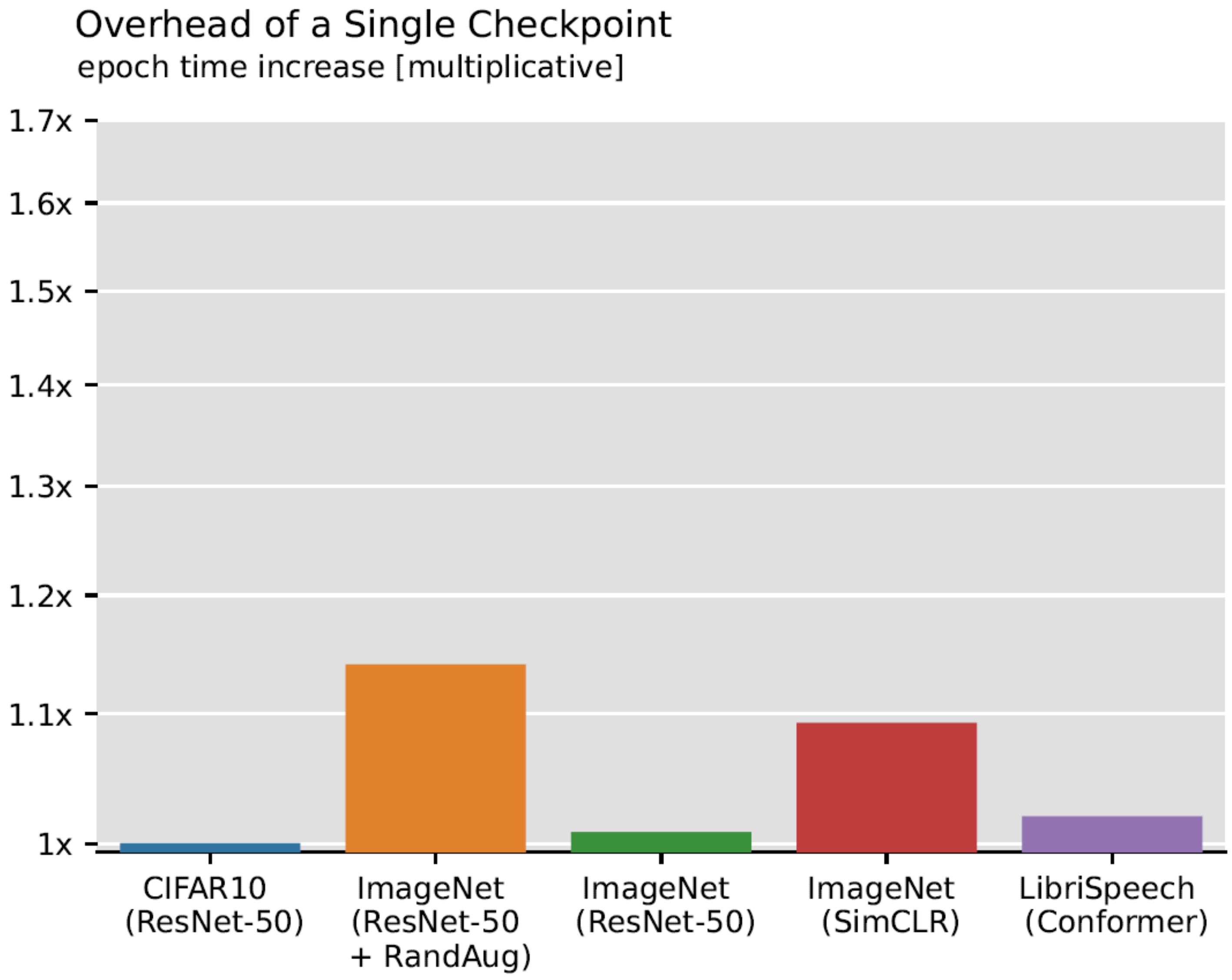
2) Worker checkpoints recover some of the progress (Req3)

Overhead Revisited

overhead = detection + failover + checkpointing + recomputation

Checkpointing Overhead

- Pipeline needs to be checkpointed in a consistent state (so it is locked)
- This will result in a drop in throughput
- Overhead is highly dependent on the structure and contents of individual pipelines
- Tradeoff between checkpointing and recomputation overhead
- Depends highly on the failure pattern



Questions

How does checkpointing overhead compare for different pipelines?

Is it feasible for all pipelines to make even just a single checkpoint?

Takeaways

1) Overhead varies with pipeline

2) Sometimes no checkpoint is better

Conclusion

- Built a system which satisfies all requirements (distributed, exactly-once, performant, bounded overhead and reproducible randomness) in the C++ layer maintaining full compatibility with all tf.data input pipelines
- Evaluation shows that worker checkpoints are not always feasible

Further Work

- Optimal checkpointing frequency
- Optimize checkpointing overhead (e.g. specific Ops)
- Use small independent sets of work (like Meta's DPP)
- Try to flush out pipelines (as in Meta's Check-N-Run)

References

Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Check-Freq: Frequent, Fine-Grained DNN checkpointing. In 19th USENIX Conference on File and Storage Technologies (FAST 21), pages 203–216. USENIX Association, February 2021.

Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. Proc. VLDB Endow.,

JCS Kadupitige, Vikram Jadhao, and Prateek Sharma. Modeling the temporally constrained preemptions of transient cloud vms. In Proceed-ings of the 29th International Symposium on High-Performance Paral-lel and Distributed Computing, HPDC ’20, page 41–52, New York, NY, USA, 2020. Association for Computing Machinery.

Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 689–706, 2022.

Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. {Check-N-Run}: a checkpointing system for training deep learning recommendation models. In 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), pages 929–943, 2022.

Convoluted Pipeline Example