[Title & Challenge (1 min)]

Hello everyone—I'm Sunjae Kim, and today I'll introduce **MC–JSim**, a new method for measuring similarity between JSON documents.

First, I'll introduce the challenge. JSON has become the universal interchange format for web services, mobile apps, and document stores—but because it has no fixed schema, two files that describe the same logical entity can differ wildly in key order, nesting structure, array ordering, or even tiny tweaks in primitive values.

Now you may wonder why we need a similarity measure for JSON. Without a reliable similarity measure, tasks like clustering JSON records by type, detecting duplicates, tracking schema drift, or finding anomalies simply can't scale: minor formatting differences or reordered fields break naïve comparisons, and exact tree-edit algorithms are too slow for large repositories.

Our goal is to match this variability with a robust similarity score that captures both structure and content, respects edit-distance bounds, and runs in near-linear time

---

[Related Work (1 ½ min)]

Next, some context in existing methods.
Traditional string metrics like Levenshtein or LCS give fine-grained edit distances but run in quadratic time and break under reordering. Line/block tools such as diff, rsync are faster but lose precision—one misplaced token can mark a whole line changed. JSON-aware tree edits such as JSON Patch or JEDI capture hierarchy but remain costly which is NP-hard. And heuristic methods often miss subtle moves or array reorderings. In short, prior work forces a trade-off between efficiency, structural fidelity, and content sensitivity.

[Problem Statement (1 ½ min)]

Then, we formalize what we need, a problem statement
We first model each JSON document as a labeled tree—nodes for objects, arrays, primitives; edges labeled by keys or indices. Edits consist of insert, delete, replace, move, and reorder, and they transform one tree into another at defined cost. Our **problem** is: given JSON documents J1 and J2, we should design a similarity score between 0 and 1 that must:

1. Drop when structure misaligns (keys, nesting, arrays).
2. Penalize primitive value changes proportionally.
3. Be Lipschitz-bounded by minimum edit cost (so small edits ⇒ small score changes).
4. Run in near-linear time on realistic JSON.

[High-Level Approach (MC–JSim) (1 min)]

To meet those requirements, we propose **MC–JSim.** At a high level, MC-JSim contains:

1. A Monte Carlo MinHash sketch for structural overlap.
2. An exact primitive comparison for content difference
3. A structural-type penalty for missing or retyped paths. One pass per document builds a fingerprint; and comparisons use only the fixed-size sketch plus lookups

---

[Step 1—Path–Token Fingerprinting (2 min)]

Next, we dive into step 1 – single-pass fingerprinting
We perform a depth-first traversal and emit a **path–token shingle** with path and encoded value at each leaf. The path is the full sequence of keys and array indices; the token encodes containers ('obj', 'arr'), rounded numbers ('num_v'), hashed strings, or booleans/null. Collecting these into a multiset S(J) captures complete hierarchy, array order, and primitive content in a compact multiset."

[Step 2—Monte Carlo MinHash Sketch (2 min)]
Moving on, we compute a **MinHash sketch**. Using r deterministic salts, we hash each shingle with MD5 algorithm and take the minimum per salt:

The sketch MH(J) approximates the set's Jaccard. Comparing two sketches by fraction of equal entries gives an unbiased Monte Carlo estimator J hat, with error decaying exponentially in r.

Here I want to point out that we consider MinHash Sketch as Monte Carlo because it uses randomized hash to approximate the true Jaccard similarity rather than computing it exactly and because it has error bound which is common for Monte Carlo patterns.

[Step 3a—Content Similarity (1 min)]
Next, the **content term** aligns on shared paths and computes per-path divergence by relative numeric error, string overlap ratio or LCS ratio, and exact boolean/null match—averaging to Sc=1−Dc. This captures semantic drift in primitive values.

[Step 3b—Structural-Type Similarity (1 min)]
Then, the **structural-type term** covers missing or retyped fields. Over the union of all paths, we penalize any absent path or type mismatch between object to array and to primitives, normalizing to Ss=1−Ds. This term registers schema-level shifts.

[Final Score & Properties (1 min)]
Afterwards, we combine them: We blend the three signals of minhash-sketch estimator J hat, content similarity S_c, and structural similarity S_s with weights α, β, γ summing to 1. This score is symmetric, in [0,1], and 1-Lipschitz w.r.t. minimum edit cost (so each atomic edit yields a bounded score change). Here I want to add that having an Lipschitz bound guarantees that the similarity score changes naturally according to the change in the JSON documents, similar to the strict conditions of being a distance metric as we

learned in class. Now complexity is one pass O(|J|) to fingerprint, and O(r) to compare.

[Experiments Overview (1 min)]
Moving forward, our experiments use 26 Compact variants of a 450-node record, 18 large variants of a 1,099-node graph and we merged them into 44 JSONs. Plus we conducted synthetic tests up to 10 k nodes for performance.

[Sensitivity & MinHash Accuracy (1.5 min)]
Now, observe the **sensitivity chart on the left**: The sensitivity chart shows tiny drops for primitive edits, large drops for array reorders, and zero for schema swaps. The **MinHash accuracy plot** on the right shows ~0.01±0.002 error—well below the 0.125 theoretical bound—on both shallow and deep JSON.

[Latency & Throughput (1.5 min)]
Following, is the performance of latency and throughput. Based on results of 100 to 10k nodes, we observe that it takes few microseconds per node for fingerprinting, 150 microseconds per node for sketching, and around milliseconds for comparison. Next, based on this we estimate for 100k nodes that fingerprinting will take ~0.24 s, sketching will take 15s, comparison will take 14s, thus confirming that it takes near-linear runtime.

[Clustering & Ablation (1 min)]
Next, clustering with a single threshold recovered intuitive groups on a 13-file set and perfectly separated 4 types in a 7-file set. In addition, the **ablation** of measuring ROC-AUC, a measure for the classifier's performance, dropped from 0.941 to ~0.80 without either content or structure similarity term, which emphasizes the importance of having all three terms.

[Key Takeaways (1 min)]
Finally, the key takeaway is that **MC–JSim** delivers:

- **Fine-grained sensitivity** (value vs. structural edits)
- **Tight probabilistic accuracy** (MinHash error ≈ 0.01)
- **Near-linear runtime** on large JSON
- **Robust clustering** with different collections

Thank you—happy to take questions!