Eval: From Computer Systems to Agents

Arun Kejariwal

As agentic deployment becomes ubiquitous - and recent security incidents [1, 2, 3] sharpen the stakes - Agent Eval design now sits alongside frontier model development.[1] An Agent Eval may assess cost, latency, correctness, groundedness, reliability, instruction adherence, reasoning, citation accuracy, relevance, bias, safety, compliance, auditability. Defining and measuring these requirements jointly makes evaluation difficult. A maintained evaluation corpus records domain requirements and failure modes, and lets teams assess model changes systematically.

Agent Eval shares a methodological foundation with Computer Systems Eval, which has been studied for decades. For example:

- *[Computer Systems]:* Is the observed performance (say, tail latency) limited by the algorithm, the underlying implementation, the generated code, the hardware or the infrastructure?
- *[Agent]:* Is the observed success[2] rate limited by model capability or by elicitation (such as system instructions, few-shot examples, chain-of-thought or explicit planning, code execution and test feedback, self-critique and retry loops)?
- *[Shared failure mode]:* A tail-latency number and a task success rate are both projections of many interacting components onto a single scalar, and neither projection is invertible. Attribution becomes guesswork, mitigations target the wrong cause, and the win measured in the harness does not survive to production.

Both involve multiple objectives - for example, tail latency and power in systems evaluation, or accuracy and cost in Agent Eval - and a combinatorial space of test conditions.

Computer Systems Eval

During an internship at Intel in 2005-6, I characterized the Core 2 microarchitecture using SPEC CPU2006[3] and VTune. Even then, the breadth of available performance events made both selection [4, 5] and interpretation hard. Interpretation was the harder half: identifying which events explained a performance limit, and therefore which compiler optimizations or next-generation hardware changes were worth pursuing. The table below illustrates the breadth of published events[4] across selected CPU architectures (refer to Appendix B for "What's uncore?").

CPU	Core event leaves	Shared/uncore leaves	Source(s)
Intel Clearwater Forest (2026)	253	342	Link
AMD Zen 6 / Venice (2026)	424	30	Link
NVIDIA Olympus / Vera (2026)	464	51	Link, Link

Event inventory size is only one part of the difficulty. The readings themselves are not as reliable as they may come across: for example, Weaver and McKee found that even retired-instruction counts varied by up to 1.07% across runs and machines, driven by interrupts and page faults, address-space randomization, and minor changes to the experimental setup reduce observed errors to less than 0.002%. Measurement validation recurs for Agent Eval, in a sharper form, once graders enter the picture later in the essay.

An extensive event inventory creates a search problem: which measurements help explain the performance of a workload? Categorizing events by subsystem structures the search, narrowing the

[1] Evaluation criteria can also supply reward signals for post-training, making independent validation essential when the system is optimized against those criteria.
[2] Different flavors of success are tabulated in Appendix A.
[3] SPEC CPU2006 has since been followed by CPU2017 and CPU2026. Successive suites refresh application workloads and resource demands to maintain relevance as software and hardware evolve; benchmark renewal reflects more than saturation of existing scores.
[4] There are many events that are not published [6].

investigation to instruction delivery, execution, address translation, caches, memory, or interconnects. Evidence of address-translation overhead, for example, can motivate testing huge pages; evidence of memory stalls can motivate investigating locality or prefetching. Further, categorization guides attribution, and attribution guides candidate optimizations - but a candidate optimization's value must ultimately be established through controlled changes and end-to-end measurement.

Hardware heterogeneity makes this harder: event definitions and availability differ across vendors, architectures, and generations, and the same top-level abstraction requires different - sometimes approximate - event derivations on each. A common categorization organizes the investigation without implying that similarly named events are interchangeable. Appendices C and D provide the CPU and accelerator breakdowns.

Counter analysis is one part of this investigation. End-to-end performance depends on interactions across the system stack, while the experimental setup can distort the gains we attribute to an optimization. Mytkowicz et al. showed that seemingly innocuous setup choices can bias performance comparisons and lead to incorrect conclusions. The table below maps the layers that can limit performance, cause latency or throughput to fluctuate, or bias their measurement.

Microarchitecture	Storage Hardware	Operating System	Language Runtime	Software Runtime
Software Application	Network Stack	System Architecture	Queueing & Scheduling	Distributed Systems*
Cloud Infrastructure	Network Infrastructure	Observability		

* This includes, for example, service aggregation, cluster orchestration, storage sharding, load balancing and consensus layer.

These effects can potentially direct optimization effort toward the wrong performance bottleneck or produce apparent gains that fail to persist under production conditions. Repeated measurements help characterize run-to-run variation[5], but this alone does not eliminate systematic bias. Appendix E describes the mechanisms within each layer. Establishing an improvement requires testing the hypothesized bottleneck and confirming that the gain survives end-to-end under representative workloads and deployment conditions.

Agent Eval shares several methodological challenges with Computer Systems Eval: workload representativeness, measurement error, attribution across interacting components, and end-to-end validation. In both, a scalar score can conceal the causes of success or failure. A microbenchmark gain need not translate into a production gain; similarly, a higher eval score need not yield a more effective deployed agent. Both require validation under representative workloads, resource budgets, and deployment conditions. Instrumentation and graders must themselves be validated, and optimization must preserve required correctness constraints while trading off preferences.

Agent Eval

An evaluation corpus is the bedrock of Agent Eval. It can be assembled through real-world data collection, synthetic generation,[6] and reuse of established benchmarks. For agents, a test case includes not only inputs and reference outputs, but also the initial environment state, available tools, permissions, resource limits, and success criteria. The dimensions below determine which conclusions the resulting evaluation can support.

Construct validity, in Messick's sense,[7] asks whether the tasks, scoring rules, and interpretation justify the capability or deployment claim. A high score on static visual questions, for example, does not by itself establish reliable visual reasoning during interaction. The Computer Systems parallel is direct: a microbenchmark speedup alone does not establish a production performance gain.

---

[5] From roughly 900,000 measurements across 835 servers over ten months, Maricq et al. found that more than 99% of configurations violated normality, and recommended non-parametric intervals, randomized run ordering, and a resampling procedure for deciding how many repetitions are enough.

[6] Fresh, representative evaluation cases can be costly to collect and annotate, especially for rare failures. Synthetic generation can expand coverage and enable controlled variations, but generated cases and reference outcomes require validation. Novel wording or imagery alone does not establish independence from training data or relevance to deployment.

[7] Jacobs and Wallach gave the computational-systems version: an evaluation operationalizes an unobservable construct, and the gap between the construct and its operationalization is where the measurement fails.

Category	Dimension	Definition
Dataset Curation[8]	Reference-error rate	Share of reference labels or verdicts found incorrect.
	Annotation disagreement	Extent of disagreement under a shared rubric.
	Coverage & stratification	Coverage of target tasks, users, environments, and risk levels.
	Population weighting	Do aggregate scores reflect the target deployment mix?
	Ungradeable-item rate	Share of items lacking sufficient evidence or criteria for a verdict.
	Input fidelity	Are task-relevant content, structure, and units preserved?
	Generator-grader dependence	Could shared model errors bias test creation and grading?
	Refresh cost	What does one cycle of re-annotation cost in expert hours, and how does that bound the cadence?
Synthetic Data	Scenario fidelity	Do generated cases preserve task-relevant real-world constraints?
	Scenario diversity	Do cases vary meaningfully across tasks and failure modes?
	Reference validity	Are generated cases and expected outcomes independently checked?
	Evaluation transfer	Do conclusions hold on independent real-world tests?
	Generation bias	Which cases or groups are systematically omitted or distorted?
Benchmarks	Contamination / leakage	Has the eval set entered training, directly or indirectly?
	Saturation / headroom	Can the benchmark still tell good from better?
	Relevance & refresh	Does the suite reflect current tasks and failure modes, and when should it be updated?
	Held-out set integrity	Has the final test set remained separate from training and tuning?

Disagreement among annotators is not automatically a defect. Voorhees established the classical result in information retrieval: assessors disagree substantially, yet relative system rankings stay nearly unchanged, which is why test collections were considered durable. That result no longer holds unconditionally - re-annotating TREC Deep Learning collections, Parry et al. found that some systems degrade substantially under fresh judgments, and that collections have a shelf life. The implication for Agent Eval is specific: disagreement rates are a weak signal on their own, and the question worth asking is whether decisions change when the labels are redone, not whether the labels match.

Training-data contamination and benchmark-specific optimization can both inflate apparent generalization, although their mechanisms differ. Benchmark renewal addresses changing workloads and failure modes as well as declining ability to distinguish systems; the interval over which a benchmark stays informative is shorter than most teams budget for.

For embodied agents, evaluation must extend to closed-loop outcomes: actions change the environment and therefore the agent's subsequent observations. Success depends on timely decisions, safe execution, and recovery from mistakes - not merely the quality of a generated answer. Simulation can broaden coverage and improve reproducibility, but its conclusions require validation against real-world behavior.

Success and Output Quality

Agent success requires achieving the intended outcome within the allowed authority and resource budget, without prohibited side effects. An agent's claim of completion must be checked against what actually

[8] Report production-weighted results separately from targeted stress tests and results by task or risk group. Annotator disagreement may reflect ambiguity or legitimate preferences rather than an incorrect reference; multiple valid outcomes do not make a task ungradeable.

happened: did the requested change take effect, was the resulting state correct, and were any constraints violated along the way? The checks below establish task success; output-quality dimensions provide complementary diagnostics. Appendix A lists additional definitions and measurements of success.

Dimension	Definition
Final-state correctness	Does the resulting environment satisfy the task's goal conditions?
Behavioral compliance	Were required behaviors completed without prohibited actions?
Budgeted success	Was the goal achieved within the specified resource limits?

An agent's output quality is multifaceted, just as a microarchitecture cannot be characterized by instructions per cycle (IPC) alone. The dimensions below provide complementary diagnostics:

Dimension	Definition
Correctness	Are facts, computations, and deliverables correct for the task?
Groundedness / faithfulness	Are evidence-dependent claims supported by the supplied evidence?
Abstention appropriateness	Does the agent decline, clarify, or escalate when - and only when - evidence or authority is insufficient?
Relevance	Does the response address the user's request?
Completeness	Are all parts of the request addressed?
Coherence	Is the response logically consistent and well structured?
Conciseness	Is detail proportionate to the task?
Instruction following	Does the response satisfy applicable instructions?
Format / schema validity	Does the output conform to the required syntax and schema?
Calibration	Across predictions, does confidence match observed correctness?
Citation validity	Do citations identify genuine, traceable sources?
Citation support	Do cited sources support their associated claims?
Citation coverage	Are claims requiring citations adequately cited?

When evidence or authority is insufficient, clarification, abstention, or escalation may be the correct outcome. This needs to be scored explicitly, because the default is perverse: a benchmark that marks abstention as failure rewards guessing, and an agent optimized against it will learn to assert rather than ask. Specify, per task, whether abstention is correct, acceptable, or a failure - and, where it is correct, whether the agent identified the right reason for it.

Operational Evaluation and Decision-Making

Operational evaluation asks whether an agent can meet the required quality, reliability, cost, and latency targets under representative deployment conditions. As in Computer Systems Eval, an improvement must be assessed against a baseline, under comparable resource constraints, and at a magnitude that matters in practice. The relevant decision may be whether to ship a change, select a model, increase an execution budget, or investigate a regression. Alongside throughput, latency percentiles, availability, and tool/API failure rates, track:

- *[Cost per successful task]* Total execution spend across all attempts divided by successful tasks. Report success rate and cost per attempted task alongside it; separate evaluation overhead unless reporting fully loaded cost.

- *[Token usage and efficiency]* Track input, output, and cached tokens, context growth, and successful tasks per token under a fixed task mix.
- *[Repeated-run reliability]* Measure per-task success across repeated trials under fixed conditions, alongside sensitivity to perturbations and the severity of failures. pass@k credits succeeding at least once, while pass^k requires succeeding every time
- *{Regression-detection delay]* Time from a regression's introduction to its detection, reported with false-alarm and missed-detection rates.

A higher success rate achieved through more retries, longer reasoning, or additional tool calls may reflect greater resource use rather than greater capability. Kapoor et al. make this the central argument for agent benchmarking: accuracy-only leaderboards reward needlessly complex and expensive agents, and the object that actually supports a decision is a cost-accuracy Pareto frontier rather than either axis alone. Compare agents at matched budgets, or report the frontier. Specify how attempts are selected, verified, and counted [7]. The deployment question is whether the additional success justifies its incremental cost and delay.

Operational requirements also depend on modality. For live audio/video, measure event-detection latency, synchronization error, interruption handling, missed-event and false-intervention rates, and deadline misses [8, 9]. For embodied systems, assess action feasibility, collision rate, control stability, and recovery under sensor noise.

Experimental Design

These metrics support decisions only when the experiment distinguishes the proposed change from differences in workloads, execution conditions, and measurement. Define the target task population, comparison baseline, resource limits, and decision criteria before collecting results. As in systems benchmarking, control relevant configuration differences and use randomized or interleaved comparisons where feasible to reduce confounding from changing conditions. The measurement framework should capture:

Element	What to record?
Test specification	User goal, initial state, applicable policy, allowed actions, success criteria, prohibited side effects and resource limits.
Execution integrity	Whether each configuration actually ran the intended tasks: completed-trial counts for every configuration compared, and the disposition of every task that did not complete - crash, timeout, API error, context-length failure, filtered.
Evaluation provenance	Where the test and reference answers came from; versions of the dataset, model, prompts, harness, tools and grader; which configuration produced the result.
Experiment setup	Serving configuration, hardware, concurrency, cache state, tool versions, and environment reset.
Execution evidence	Actual inputs, outputs, tool calls, state changes, timestamps and evidence supporting each verdict. Timeouts, exhausted budgets and tool failures with explicit categories.[9]
Failure attribution	Whether the observed failure arose from the model, tools, environment, grader, reference data - or cannot yet be determined.
Decision criteria	Required improvement, tolerated regressions, risk limits, and stopping rules.

Choose the analysis to match the sampling and dependence structure [10]:

- Plan sample size: Use the smallest practically meaningful effect, expected variation, and sampling structure to determine the required sample size (Appendix F).

[9] They count against end-to-end service quality when users experience them, but should not be silently mixed into an isolated capability score. Do not rerun failed trials until success and discard the original attempts.

- Quantify uncertainty: Report confidence or credible intervals for key metrics and differences between configurations. Use clustered standard errors when questions or tasks are related - on real evaluations these can exceed naive standard errors by more than a factor of three - and note that normal-approximation intervals are unreliable below a few hundred items, which is the regime most agent benchmarks occupy.
- Pair comparisons: Evaluate configurations on the same tasks. Pairing controls shared task difficulty and can improve precision when outcomes are positively correlated.
- Account for dependence: Use cluster-aware uncertainty estimates when tasks share documents, repositories, users, or scenario templates. Identify the independent sampling unit.
- Preserve repeated trials: Retain trial-level outcomes and distinguish within-task variation from differences between tasks. Task-level averages may still be appropriate for the analysis.

Repeatedly selecting prompts, models, or policies against the same evaluation set results in overfitting. Keep a final test set separate from development, account for multiple comparisons when making inferential claims, and pre-declare a valid sequential procedure if results will be inspected for early stopping.

From Measurement to Decision

Before running the evaluation, specify what evidence would justify shipping the candidate, rejecting it, or extending the evaluation. Define the minimum worthwhile improvement, the largest acceptable regression on other required metrics,[10] and the task or risk groups that must meet their own thresholds. Statistical significance alone does not establish practical value; an inconclusive result does not establish equivalence. To claim no regression, run an equivalence test or a non-inferiority test: the confidence interval for the difference must fall entirely inside $\pm\Delta$, where Δ is the largest acceptable regression you have already defined.

Assess the candidate jointly against these criteria. A small average improvement may be unacceptable if severe failures increase or a critical user group regresses.[11] When uncertainty spans a decision boundary, gather targeted evidence or defer the decision. When criteria are met, validate the result under representative deployment conditions.

The parallel with Computer Systems Eval is direct: component measurements explain where to investigate, controlled experiments establish which changes help, and end-to-end results determine whether those changes are worth deploying.

Trajectory

When an evaluation identifies a regression or missed target, the next question is what to change. Outcome checks establish what the agent achieved. Trajectory evaluation examines how it acted: whether it respected constraints, used tools appropriately, recovered from failures, and stopped at the right time. Trajectory evidence helps localize failures and formulate hypotheses about their causes [11]; controlled interventions are needed to test those hypotheses. For agents that retrieve evidence, retrieval diagnostics help distinguish missing or irrelevant context from errors in using that context. Context precision measures the share of retrieved items relevant to the task; context recall measures the share of required evidence retrieved. Specify the retrieval unit - document, passage, or evidence item - and the reference evidence used to estimate recall.

Retrieving relevant evidence does not ensure that the agent uses it effectively. The 'Lost in the Middle' work found that, on the evaluated tasks, model performance depended on where relevant information appeared in the context, often declining when it appeared in the middle. Varying evidence position while holding its content fixed can help distinguish failures to use available evidence from failures to retrieve it.

---

[10] For example, p99 latency, unauthorized-action rate, or success on a critical task group.

[11] Observing zero failures does not establish zero risk. With zero failures in n independent Bernoulli trials, the approximate one-sided 95% upper bound on the failure probability is 3/n. Dependence between trials or changes in deployment conditions limit this interpretation.

Additional trajectory metrics are tabulated below:

Metric	Definition
Tool selection	Was the tool suitable for the subgoal and permitted?
Argument correctness	Were arguments valid and appropriate to the intended action?
Required-call F1	Harmonic mean of call precision and recall against a justified required-call set.
Trajectory efficiency	Steps, tokens, and time relative to a valid baseline.
Plan quality	Does the stated plan address goals, constraints, and dependencies?
Error recovery	Does the agent recover from failures or stop appropriately?
Termination behavior	Does it stop on completion, futility, or budget exhaustion?
Context / memory retention	Does it preserve relevant facts and constraints across steps?
Delegation correctness	Is the subtask assigned with sufficient context and suitable authority?
Verification behavior	Does it check consequential results before relying on them?

Required-call F1 applies only when the task justifies a specific call set; otherwise, evaluate required effects and constraints. A reference trajectory provides a comparison baseline, not necessarily an optimum. Plan quality assesses an explicit plan, not an assumed account of the model's internal reasoning.

Multiple trajectories may satisfy the same task. Evaluation should therefore check required outcomes, necessary intermediate conditions, and prohibited actions without demanding a reference sequence. An alternative tool or a shorter route may be equally valid. The challenge is to distinguish harmless variation from behavior that violates the task's constraints.

- *[Trajectory coverage]* The space of tool choices, arguments, observations, and environment states is too large to enumerate. Combine production-distribution sampling with targeted tests of high-risk actions, rare failures, and recovery paths. For new products, begin with task requirements, expert-authored scenarios, and threat models, then refine coverage using observed traces.
- *[Controlled environments]* Reproducible evaluation requires known initial state and controlled dependencies. Simulators, sandboxed applications, database snapshots, and tool fixtures provide different ways to achieve this. Reset mutable state between trials and record environment versions. τ-bench checks task completion against the resulting database state; AgentDojo evaluates agents in the presence of prompt-injection attacks.
- *[Environment validity]* Simulated users and tools can introduce unrealistic behavior or omit important failure modes. Validate them against observed behavior where possible, and distinguish agent regressions from changes in the simulator. Unreset state can also leak information or alter the difficulty of later trials.
- *[Failure diagnosis]* Inject timeouts, malformed tool responses, stale state, or permission denials to test recovery and safe stopping. To investigate a suspected cause, replace one component's output while holding other conditions fixed where feasible - for example, supply verified evidence in place of retrieved context. Such interventions test a specific hypothesis; they may change the subsequent trajectory and do not establish a universal attribution.

Trajectory evaluation extends a familiar systems practice: use execution traces to locate suspicious behavior, then controlled experiments to test its cause. Agents add a challenge in specifying acceptable behavior: valid action sequences may be open-ended, and their appropriateness can depend on user intent, permissions, and changing external state. The evaluator must recognize legitimate alternatives while detecting consequential violations. As with an optimized program, taking a different route is acceptable only if the required behavior is preserved.

Rubric Design

Real-world evaluations often require nuanced, multi-criteria judgments. A rubric translates the intended outcome into explicit criteria, evidence requirements, and scoring rules. Its quality and the judge's ability to apply it must be validated separately. The following principles guide rubric design:

- Define the construct and unit of analysis. Specify what is measured - such as factual accuracy, authorized task completion, or recovery competence - and whether it is assessed per claim, response, conversation, or completed task. Generic scales underperform instance-specific criteria here: evaluators supplied with a rubric written for the particular case, plus a reference answer, track human judgment substantially better than the same model given a general quality scale.
- Identify dependencies and overlaps. Specify when criteria apply, avoid unintentionally counting the same defect twice, and distinguish diagnostic checks from criteria that determine task success or release readiness.
- Separate mandatory constraints from quality preferences. Specify which violations force failure and which dimensions permit trade-offs.
- Define pass, fail, not applicable, and insufficient evidence. For graded criteria, give examples at meaningful levels and borderline cases.
- Define aggregation explicitly. State how criteria combine and how not-applicable items and missing evidence affect scores and decisions. Missing evidence must not silently count as a pass. For example, an unauthorized external write fails the task regardless of its completeness score.
- Preserve legitimate preference differences. Distinguish ambiguous criteria from genuine disagreement, and report agreement by criterion alongside the frequency of each verdict. A pooled agreement score can conceal disagreement on rare, consequential failures.
- Validate the rubric with adjudicated examples: known defects, valid alternatives, borderline cases, adversarial wording, and missing evidence. Check whether important defects are captured and legitimate variations remain acceptable.
- Version rubric changes. Record revisions to criteria, examples, and scoring rules, and re-adjudicate affected cases before comparing results across versions [12]. Shankar et al. observed criteria drift - people refine what they are grading for by the act of grading, so the criteria are partly discovered rather than specified in advance. Versioning is therefore intrinsic to the method, not hygiene applied to it.

Rubrics can also supply training rewards. Rubrics as Rewards (RaR) uses instance-specific rubric checklists as the reward signal for on-policy reinforcement learning in domains without readily verifiable answers - a training method rather than an evaluation one, and a reason to keep the two rubrics separate.

When evaluation criteria become optimization targets, the failure is quantified rather than merely anticipated. Gao et al. showed that gold-standard performance rises and then falls as a policy is optimized further against a proxy reward. Skalse et al. showed that proxy and true objectives which cannot be gamed at all essentially do not exist for non-trivial cases. So retain independent tests for the gap between satisfying the rubric and achieving the intended goal, including deliberate attempts to exploit it - and treat a rubric used for training as compromised for evaluation of the model it trained.

Judging

Judging applies the rubric to the available evidence. Executable checks can assess properties such as schema validity, test results, and environment state; model-based judges can assess semantic and context-dependent criteria that lack a simple executable check [11, 13]. Human experts adjudicate uncertain or consequential cases. The judging protocol should specify:

- *[Independent validation]* Test judges on expert-adjudicated cases held out from rubric and judge tuning, covering relevant domains, criteria, and failure severities.

- *[Decision errors]* Measure false acceptance of failures and false rejection of valid outcomes. Validate the final acceptance or escalation rule, not only average agreement with experts.
- *[Unresolved judgments]* Permit ties in pairwise comparisons and distinguish insufficient evidence from failure or non-applicability.
- *[Escalation]* Validate confidence- or disagreement-based escalation on held-out cases. Report consequential errors left unreviewed alongside the human-review workload.
- *[Evidence]* Preserve the passages, tool results, and state records supporting each verdict, with exact references. The grader must also have access to the evidence required for its verdict.
- *[Reproducibility]* Record judge, prompt, rubric, tool, and aggregation versions; revalidate after material changes.

Sources of error include:

- *[Insufficient expertise]* The judge understands the criterion but cannot reliably assess the underlying facts or reasoning [14].
- *[Context loss]* Truncation, summarization, or long traces obscure decisive evidence.
- *[Systematic bias]* Scores change with presentation order, verbosity, or other features unrelated to the criterion. Self-preference is the case least amenable to prompt engineering: a judge's tendency to favor its own output is causally linked to its ability to recognize that output as its own [15], which argues for separating judge and candidate model families rather than instructing the judge to be impartial.
- *[Injection]* Instructions embedded in evaluated content attempt to redirect the judge or alter its verdict. This is an optimizable attack, not only an opportunistic one: gradient-optimized sequences appended to a candidate response reliably induce a judge to select it (JudgeDeceiver), outperforming handcrafted injections and surviving known defenses.

Multiple judges may reduce individual errors, but agreement does not establish correctness: judges can share blind spots, training influences, and responses to superficial cues. Validate the ensemble's final decision rule on independent cases. As with profilers, repeated measurements or agreement among instruments do not eliminate systematic measurement error.

Distinguish the agent's compliance with safety constraints, the judge's resistance to injected instructions, and the evaluation environment's containment of consequential actions. Each requires separate checks; Appendix G summarizes the relevant safety dimensions.

Meta-Eval

Judge validation is one part of meta-evaluation: assessing whether the complete evaluation provides trustworthy evidence for its intended use. Two questions must be answered separately:

- Is the evaluation internally sound? Do the rubric, judge, and measurement procedure detect important failures, accept valid alternatives, and quantify uncertainty appropriately? Agreement with expert-adjudicated references is useful evidence, but those references also require scrutiny.
- Is the evaluation externally valid? Do its conclusions hold for the tasks, users, environments, and outcomes relevant to deployment?

Reliability alone is insufficient - this is the reliability-versus-validity distinction from measurement theory [N9, N10], and it is why a judge can be consistently wrong. A study of 21 LLM judges found test–retest reliability above 0.95 alongside substantial position bias under the tested protocol. JudgeBench found that several strong judges performed only slightly above chance on challenging correctness comparisons. Chance-corrected measures collapse when one verdict dominates, which is the kappa paradox: high observed agreement with near-zero kappa under skewed prevalence or rater bias. Report raw agreement, a chance-corrected measure, and the marginal distribution together, and prefer estimators built for high-agreement or skewed regimes [16, 17] over a single headline number. EvalGen showed that

reviewing outputs can change how people interpret evaluation criteria, motivating rubric versioning and re-adjudication when criteria change.

The Computer Systems Eval parallel is workload representativeness: a benchmark result supports a deployment decision only to the extent that its workload and execution conditions represent what matters in production. Agent Eval inherits this requirement and, when model-based judges are used, adds another dependency: the scoring mechanism must itself generalize to the deployment domain.

Validate offline conclusions against deployment outcomes: successful task completion, human effort and rework, completion time, failure severity, and user satisfaction. Interpret proxies in context. Fewer human interventions may indicate useful autonomy or missed escalation; fewer edits may conceal an undetected error. Retention and business metrics provide downstream evidence, but attributing changes to the agent requires a suitable comparison, ideally randomized where feasible.

Revalidate after material changes to the agent, judge, rubric, tools, or task distribution, alongside periodic audits. Retain an overlapping anchor set to help distinguish changes in the evaluator from changes in the agent, while adding fresh cases to test current conditions. Evaluation quality must be maintained as deliberately as the system it measures.

Looking Forward

The next challenge is to make evaluation more diagnostic, more efficient, and more predictive of deployment outcomes. Four problems remain central:

- *[Attribution in changing environments]* A failure may arise from the model, retrieved evidence, tools, permissions, or external state - and from their interactions. Controlled interventions can isolate specific causes, but reproducing the relevant conditions becomes harder as agents and environments adapt.
- *[Evaluation within decision budgets]* Tasks, policies, modalities, environments, repeated trials, and runtime configurations create a combinatorial test space. The challenge is to select tests that resolve the decision at hand without overlooking rare, consequential failures. Fast regression suites, targeted investigations, and broader periodic evaluations serve different purposes.

  Systems profiling bounds an equivalent space with a stopping rule - refine only the branch carrying the mass - but that rule is sound there because the top-level categories partition a conserved quantity [18]. The open question for Agent Eval is what plays that role.
- *[Transfer from offline scores to deployed value]* Evaluation must establish when measured gains predict successful, safe, and useful behavior in production. This requires testing across changing conditions, recognizing multiple valid outcomes, and measuring human effort and downstream consequences.
- *[Coverage as a first-class artifact]* A coverage map can organize this effort: associate each test with the capabilities, risks, user groups, and environments it exercises, the decision it supports, and its known blind spots. Like subsystem categorization in systems profiling, this structure guides investigation. It does not establish coverage by itself; the remaining problem is choosing which combinations to test and how much evidence each requires.

Closing

Computer Systems Eval and Agent Eval share a central challenge: turning measurements of interacting components into trustworthy decisions. The transferable discipline is clear: combine component diagnostics with end-to-end checks, validate the measuring apparatus, preserve required behavior, and confirm that improvements survive deployment.

Agents add uncertainty about what success means, whether the grader is right, and how behavior changes during interaction. As agents take on consequential work, evaluation becomes essential infrastructure for deciding what to optimize, what to ship, and how much autonomy to grant.

A maintained evaluation system can become a moat: domain-specific cases, adjudicated failures, validated graders, and production feedback encode hard-won knowledge about what works and what breaks. Models will change. The advantage lies in knowing whether each change makes the product better.

Agent capability creates possibilities. Evaluation supplies the evidence to act on them.

Appendix

A. Agent Success

Flavor of success	Concrete measurement
Correct final answer	Fraction of tasks whose final answer matches a reference, using explicitly defined normalization or tolerances
Correct final environment state	Fraction of tasks for which the resulting database, application or filesystem satisfies the specified goal predicates
Functional correctness under execution	Fraction of tasks whose submitted implementation passes the required executable tests, including designated regression checks
Deliverable meets explicit acceptance criteria	Criterion pass rate; alternatively, fraction of tasks exceeding a predefined rubric threshold
Measurable progress toward the goal	Proportion of annotated subgoals achieved, or a defined state-to-goal matching score
Correct tool use	Fraction of tool-use cases with the correct function and acceptable arguments or execution results; include appropriate abstention when no tool applies
Required behavior achieved without forbidden actions	Required milestones satisfied **and** no prohibited event observed in the trajectory
Useful and secure behavior under attack	Measure legitimate-task completion under attack and attacker-goal achievement separately. A stricter joint criterion is: legitimate task succeeds **and** attacker goal fails
Consistent success across repeated attempts	**pass^k**: probability of succeeding on all k repeated trials of a task, aggregated across tasks
Success within a resource budget	Task-success rate at a fixed dollar, token, tool-call or time budget; compare success–cost tradeoffs

B. What's Uncore?

Uncore[12] *refers to processor components that are outside the individual CPU execution cores but reside on - or closely integrate with - the processor package. Examples include:*

- Last-level cache, such as L3/LLC
- Memory controllers
- On-chip mesh or interconnect fabric
- Inter-socket links such as Intel UPI
- PCIe and CXL controllers
- Power-management units

The distinction also determines scope:

- *Core counter:* normally measures activity attributed to one core or hardware thread.
- *Uncore counter:* normally measures an entire shared hardware block, socket, die, cache slice or memory channel. It may aggregate traffic generated by many cores and devices.

C. Categorization of Core and Uncore Events

Category	Intel Clearwater Forest	AMD Venice	NVIDIA Vera
Instruction fetch / decode	19	43	57
Branch prediction	34	6	11
Cache / prefetch	235	71	140
Memory / load-store	167	90	26
TLB / translation	21	21	41
Execution / retirement	60	223	134
Pipeline / stalls	14	0	33
System / power / interrupt	10	0	23
Interconnect / I/O	35	0	39
Sampling / trace configuration	0	0	11

[12] "Uncore" is Intel's conventional term. AMD commonly uses names such as L3, Data Fabric and UMC PMUs, while NVIDIA uses block-specific terms such as UCF, C2C and memory-latency PMUs. They serve the same broad role.

Category	Intel Clearwater Forest	AMD Venice	NVIDIA Vera
LLC / home-agent / coherent fabric	188	17 L3	20 UCF
Memory controller or memory-latency PMU	66	13 UMC	3 CMEM-latency
Socket / chip / interconnect fabric	12	-	17 C2C/C-Link/D-Link
I/O, PCIe and CXL	72	-	11 PCIe/PCIe-target
Power / system	4	-	-

In the table above, '-' means not separately identifiable in the published taxonomy, not necessarily that the hardware has no such counters.

D. Accelerator Events and Categorization

Event count across vendors

Accelerator	Low-level event leaves	Source(s)
NVIDIA Blackwell Ultra B300 (2025)	3774	Link
AMD Instinct MI355X / gfx950 (2025)	428	Link
Google TPU7x / Ironwood (2025)	2933	Link

Event Categorization

Normalized category	NVIDIA Blackwell Ultra B300	AMD Instinct MI355X	Google TPU7x
Compute / execution engines	SM/execution: 765	Compute/instruction/ wavefront: 139	TensorCore: 285 + SparseCore: 1,837
Frontend / dispatch / global control	Frontend/global: 61	Dispatch/control: 126	-
Local cache / shared memory / address path	L1/texture/shared memory: 422	L1/vector/address: 96	-
L2 / cache / external-memory path	L2/cache/control: 2,450 + DRAM/HBM: 21	L2/HBM path: 67	HBM/memory: 768
Fabric / I/O	Fabric/I/O: 55	-	Fabric/global: 35
Power management	-	-	Power management: 8

As above, '-' means not separately identifiable in the published taxonomy, not necessarily that the hardware has no such counters.

E. Sources of Performance Limits and Measurement Error

System Domain	Source / Mechanism	Relevant Layer / Subsystem	Description
Microarchitecture	Line-fill buffer exhaustion	Cache miss tracking	Finite resources for tracking outstanding cache misses can fill, stalling new memory requests and limiting memory-level parallelism.
	Reorder-buffer stalls	Out-of-order execution	A full reorder buffer limits further instruction allocation while older instructions await completion, restricting the independent work available to hide latency.
	Shared interconnect congestion	On-chip interconnect	Concurrent requests compete for on-chip fabric links, queues, and arbitration resources. Congestion increases access latency to caches, memory controllers, and I/O blocks.
	PCIe protocol and transaction overhead	PCIe host interface	PCIe transaction overhead reduces useful throughput, especially for small transfers. Transfer size, batching, and device or driver behavior determine the resulting latency and bandwidth efficiency.
	CPU idle-state exit latency	Power management	Returning from a CPU idle state delays execution. Exit latency depends on the selected core or package state and the platform. Deeper idle states generally require more time to resume work.
	Frequency transitions and throttling	CPU frequency and power control	Frequency ramp-up can delay burst processing after idle periods. Thermal or power limits can separately reduce sustained frequency. Both effects depend on policy, workload, and platform.
	NUMA remote-memory access	Memory placement and interconnect	Access to memory attached to another NUMA node can add interconnect latency and consume remote bandwidth. Placement, access patterns, and concurrent traffic determine the performance impact.
	Cache capacity and coherence contention	Cache hierarchy	Competing working sets cause capacity or conflict evictions. Separately, writes to shared cache lines cause coherence traffic and invalidations, including false sharing between otherwise independent data.
Storage Hardware	SSD garbage collection and wear leveling	Flash translation layer	Internal flash reclamation and data movement compete with foreground I/O for controller and flash resources. Resulting latency increases depend on device design, workload, free space, and overprovisioning.
Operating System	Task context switching	OS scheduler and address spaces	Switching between runnable tasks incurs scheduler and execution-state management costs and can disrupt cache or translation locality. These costs depend on the architecture, address spaces, and workload.
	OS scheduling policy	CPU scheduling	The OS scheduler allocates CPU time among runnable tasks. Fairness rules, priorities, and preemption affect request latency. Behavior depends on the kernel version and scheduling class.
	Interrupt processing	Interrupt subsystem	Device interrupts and interprocessor interrupts (IPIs) consume CPU time and may interrupt application work. Interrupt placement, frequency, and deferred processing can affect locality and tail latency.
	Background OS activity	System services	Daemons, logging, maintenance, and monitoring compete with application work for CPU, memory, storage, or network resources. Their timing and resource demand can introduce performance variability.
	Software I/O queueing	Kernel I/O path	Queue depth, I/O scheduling, locking, and completion processing can add delay or limit throughput. The effect depends on the kernel path, device characteristics, workload, and configuration.
Language Runtime	Garbage-collection pauses and contention	Runtime memory management	Stop-the-world phases pause application threads. Concurrent collection reduces some pauses but competes for CPU and memory bandwidth. Allocation rate, heap size, and collector design determine the impact.
Software Runtime	Thread lock contention	Synchronization	Threads competing for locks may block or spin. Critical-section duration, lock granularity, scheduling, and cache-line sharing determine the resulting stalls and scalability limits.
Software Application	RPC serialization overhead	Encoding and decoding	Encoding and decoding structured messages consume CPU time and may allocate or copy memory. Cost depends on the format, message size, schema, and implementation.
	Static connection-to-worker mapping	Work distribution	Binding connections to dedicated workers can produce uneven queues when connection workloads differ. Idle workers may be unable to help busy workers without work sharing or reassignment.
	Blocking I/O variability	Storage and socket access	Synchronous file or socket operations occupy a worker while waiting for data or completion. Variable service times and limited worker capacity can propagate delays to queued requests.
Network Stack	Kernel network-stack overhead	Packet processing	Protocol processing, data copies, buffer management, synchronization, and reassembly consume CPU and memory resources. Offloads, batching, and the selected I/O path change these costs.
	Packet batching and excessive buffering	Network queues and coalescing	Batching or interrupt coalescing can delay individual packets while improving processing efficiency. Excessive queue occupancy separately causes bufferbloat and sustained latency under load.
Network Infrastructure	Switch queue congestion	Datacenter network fabric	Microbursts and incast can fill switch queues, increasing queueing delay. Packet drops or congestion-control responses can add retransmission and recovery delays, depending on transport and configuration.
System Architecture	Serial microservice dependencies	Service dependency path	End-to-end latency includes the sum of serial stage latencies and communication delays. Variability depends on stage variances and their correlations. Queueing and fan-out introduce additional effects.
Queueing & Scheduling	FCFS scheduling trade-offs	Queue discipline	First-come, first-served scheduling can leave short jobs waiting behind long jobs. Its mean and tail response times relative to other policies depend on load, job-size distribution, and the optimization objective.
	Unknown job sizes	Scheduling information	When service times are unknown in advance, scheduling uses available information such as service already received or predicted job duration. Limited information and prediction errors affect latency and fairness.
	Priority-based preemption	Preemptive scheduling	A newly eligible higher-priority job may interrupt a running job. This improves some jobs' response times while delaying others, and can add context-switching costs or starvation risk.
	Delay from later-arriving work	Queue ordering	Under policies that permit overtaking, jobs arriving later can be served before a target job. Their execution extends its waiting or completion time. The effect depends on the scheduling rule and arrivals.
	Head-of-line blocking	Request queues	A request at the front of a queue can delay later requests when service cannot bypass it. The impact depends on queue organization and whether requests share the same constrained resource.
	Queueing near capacity	Service capacity	As offered load approaches available capacity, queues and response times can grow sharply. The relationship depends on arrival and service-time variability, scheduling, and admission control.
	Bursty arrivals	Workload arrival process	Short bursts can temporarily exceed service capacity and build queues, even when average load is moderate. Burst size, duration, and correlation determine the resulting latency.
Distributed Systems	Fan-out synchronization	Request aggregation	A request that requires all parallel sub-results completes only after its slowest required branch. Quorum, partial-result, timeout, or hedged-request policies change this dependency.
	Straggler exposure with fan-out	Distributed request execution	When a request depends on more components, its chance of encountering a straggler can increase. The effect depends on request fan-out, component dependence, and completion rules, rather than cluster size alone.
	Partition hotspots	Data sharding	Skewed access patterns overload particular shards or keys while other resources remain underused. Hotspots cause queueing and can limit overall throughput.
	Stale load-balancing decisions	Request routing	Routing based on delayed or incomplete load information can send work to already busy nodes. Feedback delays and synchronized decisions can create imbalance or oscillation.
	Replication acknowledgment delays	Replication and consensus	Writes wait for the acknowledgments and durability conditions required by the protocol. Majority-quorum replication generally need not wait for every replica. Leader, network, and storage delays affect completion.
	Repair and reconstruction traffic	Storage recovery	Replication repair, rebalancing, and erasure-code reconstruction consume storage, network, and CPU resources. Recovery can increase foreground latency unless resources or rates are controlled.
Cloud Infrastructure	Multi-tenant resource contention	Shared host resources	Co-located virtual machines or containers compete for CPU time, cache, memory bandwidth, storage, and networking. Isolation and resource controls determine the resulting interference.
	Resource reclamation and resizing	Cluster resource management	Reclaiming capacity from background workloads can cause preemption, migration, cache warm-up, or memory-reclaim costs. Effects depend on the mechanism. Throttling alone does not imply swapping.
Observability	Profiling observer effect	Instrumentation	Tracing, sampling, and metric collection consume resources and can change execution timing or locality. Measure collection overhead and check whether instrumentation alters the behavior being studied.
	Experimental layout bias	Program and process layout	Link order, code or data placement, and environment size can alter cache and address-layout behavior. Systematic setup differences may bias measured performance comparisons.
	Distributed clock misalignment	Trace timestamps	Clock offset, drift, and synchronization errors distort cross-host event ordering and inferred durations. Monotonic clocks support same-host interval measurement but do not synchronize timestamps across hosts.
	Coordinated omission	Load generation and latency measurement	A closed-loop load generator reduces new arrivals while responses are slow. It can understate latency for an independently arriving workload by omitting requests that would have arrived during stalls.

F. Eval Sizing

Sample size requirements vary dramatically depending on whether the metric is binary or continuous and whether the design is paired or unpaired, clustered or unclustered. Examples of each design choice follow:

- Metric Scale
  - *[Binary]* Goal completion, task pass/fail (0/1), tool-call execution success.
  - *[Continuous]* API token/dollar cost, continuous reward scores, execution latency, or multi-criteria rubric quality scores (0-100).
- Pairedness
  - *[Unpaired]* Evaluating Agent A and Agent B on separate, non-overlapping task samples (e.g., routing traffic in live A/B production tests).

    - *[Paired]* Running candidate Agent A and baseline Agent B on the *exact same* set of benchmark tasks (standard practice for offline agent benchmarking).
- Clustering
    - *[Unclustered]* Tasks/prompts are completely independent and randomly sampled.
    - *[Clustered]* Tasks are nested within environments (e.g., WebArena tasks clustered by web domain, SWE-bench tasks clustered by GitHub repository, or multi-turn trajectory steps clustered within a single dialogue session).

The table below gives large-sample planning approximations for differences in proportions or means. It does not directly size experiments on p99 latency or rare harms. Binary paired rows use Wald approximations; sparse discordant outcomes, unequal or overlapping clusters, and other departures from the stated assumptions require separate treatment.

Combination	**Formula for Sample Size (n)**	**Assumptions**
Binary, Unpaired, Unclustered	$n \approx \frac{\left(z_{1-\alpha/2}\sqrt{2\bar{p}(1-\bar{p})} + z_{1-\beta}\sqrt{p_1(1-p_1)+p_2(1-p_2)}\right)^2}{\delta^2}$	Equal n per agent; independent tasks.
Binary, Unpaired, Clustered	$n \approx \frac{\left(z_{1-\alpha/2}\sqrt{2\bar{p}(1-\bar{p})} + z_{1-\beta}\sqrt{p_1(1-p_1)+p_2(1-p_2)}\right)^2}{\delta^2} \cdot \left[1+(m-1)\rho_{cl}\right]$	Equal n per agent; common outcome ICC.
Binary, Paired, Unclustered	$n \approx \frac{(z_{1-\alpha/2}+z_{1-\beta})^2\sigma_D^2}{\delta^2}$	Independent pairs; Wald test.
Binary, Paired, Clustered	$n \approx \frac{(z_{1-\alpha/2}+z_{1-\beta})^2\sigma_D^2}{\delta^2} \cdot \left[1+(m-1)\rho_{cl,D}\right]$	Wald test; ICC of paired differences.
Continuous, Unpaired, Unclustered	$n \approx \frac{(z_{1-\alpha/2}+z_{1-\beta})^2(\sigma_A^2+\sigma_B^2)}{\delta^2}$	Equal n per agent; independent tasks.
Continuous, Unpaired, Clustered	$n \approx \frac{(z_{1-\alpha/2}+z_{1-\beta})^2(\sigma_A^2+\sigma_B^2)}{\delta^2} \cdot \left[1+(m-1)\rho_{cl}\right]$	Equal n per agent; common outcome ICC.
Continuous, Paired, Unclustered	$n \approx \frac{(z_{1-\alpha/2}+z_{1-\beta})^2(\sigma_A^2+\sigma_B^2-2\rho_{AB}\sigma_A\sigma_B)}{\delta^2}$	Independent pairs; general variances.
Continuous, Paired, Clustered	$n \approx \frac{(z_{1-\alpha/2}+z_{1-\beta})^2(\sigma_A^2+\sigma_B^2-2\rho_{AB}\sigma_A\sigma_B)}{\delta^2} \cdot \left[1+(m-1)\rho_{cl,D}\right]$	ICC of paired differences; general variances.

where,

- n: Observations per agent (unpaired) or paired tasks (paired), not clusters; round up, to whole clusters where applicable.
- p_1, p_2: Anticipated binary success rates for Agent A and Agent B; $\bar{p} = (p_1 + p_2)/2$.
- μ_A, μ_B: Anticipated mean continuous scores for Agent A and Agent B.
- δ: Target absolute difference, $|p_1 - p_2|$ for binary outcomes or $|\mu_A - \mu_B|$ for continuous outcomes.
- σ_A^2, σ_B^2: Marginal continuous-outcome variances; σ_A, σ_B are the corresponding standard deviations.
- ρ_{AB}: Correlation between agents' outcomes on the same task (binary correctness or continuous scores).
- $D_{ci} = Y_{A,ci} - Y_{B,ci}$: Paired outcome difference on task i in cluster c; omit c for unclustered tasks.
- σ_D^2: Binary difference variance, $p_1(1-p_1) + p_2(1-p_2) - 2\rho_{AB}\sqrt{p_1(1-p_1)p_2(1-p_2)}$; binary correlations must be feasible for p_1, p_2.
- σ^2: Common marginal variance if $\sigma_A^2 = \sigma_B^2 = \sigma^2$; continuous difference variance then reduces to $2\sigma^2(1-\rho_{AB})$.
- m: Common cluster size (tasks per agent for unpaired designs; paired tasks per cluster for paired designs).
- ρ_{cl}: Within-cluster outcome ICC, assumed common across agents in the unpaired clustered rows.
- $\rho_{cl,D} = \mathrm{Corr}(D_{ci}, D_{cj})$, $i \neq j$: Within-cluster ICC of paired differences; not the individual agents' outcome ICC.
- $z_{1-\alpha/2}, z_{1-\beta}$: Standard normal quantiles for two-sided significance α and power $1-\beta$.
- All rows: Large-sample approximations for testing zero difference; adequate sample size for normal inference.
- Clustered rows: Equal m; common within-cluster correlation; independent clusters (also across agents if unpaired); enough clusters for normal inference.

G. Eval Safety

Dimension	Definition
Harmfulness	Content-policy violations across a harm taxonomy.
Bias and fairness	Differential performance across demographic groups.
Jailbreak robustness	Resistance to direct adversarial prompting.
Prompt-injection robustness	Resistance to indirect attacks via tool output, retrieved documents, web pages.
Robustness to distribution shift	Does performance hold under changes in inputs or operating conditions?
Data leakage / PII	Exfiltration through outputs, tool calls or logs.
Permission and scope adherence	Does the agent stay within granted authority?
Blast radius / irreversibility	How bad is a failure, weighted by reversibility?
Deception	Does the agent misrepresent facts, actions, or intent?
Self-preservation	Does it resist authorized shutdown or replacement?
Power-seeking	Does it seek unnecessary authority or resources?
Reward hacking	Does it satisfy the checker without satisfying the goal?

The following judge-bias categories follow CALM; detection protocols are adapted for this discussion.

Bias	Mechanism	Detection protocol
Position	Favors an option by its position in the prompt.	Re-score the same pair with the two candidates swapped.
Verbosity	Prefers longer responses irrespective of quality.	Score quality-matched pairs that differ only in length (padded vs terse).
Self-enhancement	Favors output the judge's own model family produced.	Cross-family scoring: same candidates judged by models from different families.
Bandwagon	Follows a stated majority opinion.	Inject a claimed majority verdict ('80% of reviewers preferred A') into half the prompts.
Authority	Over-credits citations - including fabricated ones.	Score matched answers with and without appended references, some deliberately fabricated.
Distraction	Overweights irrelevant detail.	Append task-irrelevant but plausible content to one candidate.
Fallacy-oversight	Accepts a correct final answer despite invalid reasoning.	Score answers with correct conclusions reached via deliberately invalid reasoning chains.
Sentiment	Prefers particular emotional registers.	Score semantically equivalent answers rewritten in positive, neutral and negative registers.
Chain-of-thought	Score shifts with the presence of reasoning instructions.	Score identical candidates with and without a reasoning instruction in the judge prompt.
Compassion-fade	Scores differ when model identity is revealed vs anonymized.	Score the same candidates with attributed and anonymized model labels.
Refinement-aware	Scores differ when told an answer was refined.	Score identical candidates with and without a 'this is a revised answer' preamble.
Diversity	Differential treatment by demographic content.	Counterfactual perturbation: swap demographic markers in otherwise identical candidates.