

AI Agent Safety & Reliability Platform – Startup Proposal

Proposed Startup Idea

We propose a platform that **ensures autonomous AI agents are both reliable and safe in their tool use**. This system combines *reliability modeling* with *active safety verification* to give organizations confidence in agents that can write code, execute tools, or manage data. The core innovation is an **adaptive verification loop** that continuously maps out an agent’s “failure boundary” – the conditions under which the agent might produce errors or unsafe actions – and intervenes or informs the user before things go wrong. In practice, this means two integrated capabilities:

- **Predictive Reliability Scoring:** The platform learns a **committor function** for each agent or task – essentially the probability that the agent will succeed at a given task before failing. For example, given an agent’s current state (its plan, context, or code changes pending), the system estimates the probability of a correct outcome vs. a failure. If an agent-generated code patch has a very low predicted success probability, the platform flags it or filters it out preemptively. This saves costly trial-and-error runs and prevents obvious failures from ever executing. In essence, the committor serves as a **universal reliability metric** for agent behavior in high-dimensional scenarios.
- **Active Safety “Red Teaming”:** Simultaneously, the platform runs an **adaptive adversarial testing loop** against the agent to probe its safety. Rather than relying on static harmful prompt lists or one-off pen-tests, it uses an *active learning approach* to automatically discover “**semantic jailbreaks**” and **unsafe tool use sequences** that could exploit the agent. A dedicated “Attacker” module generates adversarial scenarios (malicious prompts, tool misuse sequences, etc.) and a “Judge” module evaluates if the agent fails (e.g. produces a toxic output or unauthorized action). Crucially, each attack is chosen based on *epistemic uncertainty*: the system focuses on borderline cases where it isn’t sure if the agent will refuse or comply, rather than repeating known safe or known unsafe inputs. Over time this creates a topological map of the agent’s vulnerability space, clustering known failure modes and **probing into nearby unknown regions of high uncertainty to find new ones**. This closed-loop process reveals complex multi-step failure modes that static benchmarks miss, such as indirect prompt injections or tool misuse that only occur in certain contexts.

Behind the scenes, these two capabilities share a common technical framework: an **Adaptive Epistemic Discovery Loop**. This loop uses an **Epistemic Neural Network (ENN)** to model

what the agent *doesn't yet know* about its success/failure boundary, and a smart sampler (e.g. Bayesian criteria like BICEP) to query the “ground truth” (by running the agent in a sandbox or simulator) only on the most informative cases. By targeting areas where the predicted success probability $q(x)$ is around 0.5 and model uncertainty is high, the system efficiently learns the dividing line between safe and unsafe (or success and failure) behaviors. This makes testing far more efficient than brute-force fuzzing: rather than waste compute on millions of trivial cases (which are almost always safe) or known extreme bad cases, it hones in on the critical *edge cases*. As a result, the platform can discover high-severity failures with orders of magnitude fewer queries than random testing, **saving time and cost**.

Finally, the platform can optionally provide **cryptographic verification (“receipts”)** for agent actions to enhance trust. Every tool invocation or external action by the agent can be recorded with a cryptographic signature or proof, creating an audit trail that the action was authorized and not tampered with. This approach, inspired by emerging “zero-trust” agent frameworks, **ensures the agent only acts on inputs that are authentic and authorized**. For example, if the agent is instructed to execute a database query, the instruction can be verified as coming from a legitimate source and the query results signed. These *cryptographic receipts* pair with the reliability/safety modeling to give end-users and regulators high confidence: not only is the agent *tested* to be robust, but every runtime action is *verifiable* and auditable.

In summary, the startup would deliver a comprehensive **AI Agent Safety & Reliability Platform**. It proactively **models and bounds the risk** of deploying autonomous agents by (a) predicting and preventing likely failures, (b) systematically uncovering novel failure modes (with a focus on tool-use safety boundaries), and (c) establishing trust through verification and audit trails. This de-risks the adoption of powerful agentic AI in real-world applications.

Market Wedge and Early Adopter Profile

Our initial wedge is to target AI systems where failure is **both costly and frequent enough to be a pain point**. A prime example is the emerging class of **LLM-powered coding agents** and co-pilots. These agents propose code changes or fixes, but running their suggestions through full test suites or CI/CD pipelines is expensive (both in cloud compute and developer time). Early trials (e.g. SWE-bench for code-fixing agents) show that success rates are only ~20%, meaning 80% of test executions are wasted on bad outputs. Our platform can immediately save costs here by **filtering out low-probability “dead-on-arrival” attempts** – if the predicted $q(x)$ of success for a given code patch is near 0, the system advises skipping the full test run. This lets dev teams focus their expensive verification on the 20% of cases with promise, effectively doubling efficiency. Early adopters in this profile are likely **software companies integrating AI into their DevOps or QA** processes (e.g. a GitHub Copilot extension for auto-fixing pull requests, or enterprises using GPT-4 to refactor legacy code). They feel the pain of unreliable agent outputs as cloud bills and slow iteration cycles, so they will pay for a solution that improves throughput.

Another strong early-adopter profile is organizations deploying **LLM agents with tool access in high-stakes environments** (finance, healthcare, enterprise automation). For instance, a bank building a chatbot that can execute transactions, or a healthcare AI that can call internal databases. Such players have a *low tolerance for unsafe behavior*: a single unauthorized action or security lapse could be catastrophic. Our platform's ability to **actively “red team” an agent's tool-use and uncover hidden failure modes** is extremely valuable here. We move beyond standard prompt filtering to simulate complex attack scenarios an agent might face – e.g. a malicious document that poisons the agent's context, or a sequence of benign-looking tool calls that together exfiltrate data. Early adopters in this category are **enterprises and SaaS providers adding agentic AI features** (for example, an enterprise software suite adding an “AI assistant” that can update customer records via an API). They need assurance that this assistant won't inadvertently violate security or compliance. Our system offers them a proactive safety net: a way to *demonstrate* and *continuously ensure* that the agent stays within safe boundaries.

In both cases, the wedge is providing **tangible risk reduction and cost savings immediately**. We are not asking customers to overhaul their AI models or pipelines – our platform slots in as a testing and guardrail layer around existing agents. This ease of integration is key for early adoption. We would offer simple APIs or a dashboard: e.g. submit an agent's action for a reliability score, or run the agent in our sandbox to get a “safety report” of potential exploits. By starting with pain points that are acute (wasted compute, security fears) and in domains where buyers have budget (developer tooling and enterprise security), we carve out a viable entry market.

Why Now

1. Agents are becoming practical and widespread: Within the last 1-2 years, large language models have evolved from just generating text to being “**agentic**” – **perceiving, deciding, and acting through tools**. OpenAI, Anthropic, and others are deploying API functions, plugins, and autonomous agent frameworks that let AI systems execute code, make web requests, or control real-world systems. This is a *sea change* in capability that also introduces new risks. As noted in our research, “*the emergence of ‘Agentic AI’ ... introduced a new and far more dangerous category of risk: State-Based Vulnerability.*” Unlike a chatbot that could only produce a toxic sentence, an agent can actually perform harmful operations (even if its output text looks benign). The industry is only beginning to grapple with this. **The timing is perfect for a solution addressing these agent-specific risks**, as more companies move from experimental demos to real deployments of tool-using AI. Everyone is asking how to trust these agents in production.

2. Limitations of current safety approaches: The status quo for AI safety and verification is lagging behind this new paradigm. Most existing tools focus on content filtering or static red-teaming with known prompts. These are insufficient for agents that operate in **dynamic, context-rich environments**. For example, companies rely on benchmarks like JailbreakBench or a set of “forbidden prompt” tests. But as our analysis showed, *static benchmarks lack*

contextual awareness (they don't cover multi-turn or tool-specific scenarios) and models quickly overfit to them, leaving blind spots. Moreover, reliability is largely unaddressed – developers currently find out an agent is unreliable only after it fails repeatedly in practice. **Recent advancements in research (including the attached frameworks) provide a new way:** using active learning and uncertainty quantification to *systematically probe an AI's weaknesses*. Techniques like the committor-based active sampling have matured in research contexts just as the need for them becomes dire in industry. This convergence of a pressing problem and an available technical solution makes now an ideal moment to build a product.

3. Technology tailwinds: Key components of our platform are now feasible where they weren't a few years ago. High-quality embedding models and vector databases (FAISS, etc.) allow us to map large semantic spaces of prompts or states efficiently – enabling the kind of graph-based uncertainty modeling we use. Open-source safety evaluators (like Llama Guard 3 mentioned in our docs) are emerging, meaning we can build a "Judge" without needing a proprietary service. There's also a cultural shift towards *verification and audit in AI*: initiatives in "model cards," "AI audit trails," and even cryptographic verification of AI outputs (as seen with companies like Crittora adding trust layers) show that the market is ready to value transparency and proof in AI interactions. All these factors lower the barrier to entry for our solution and increase the appetite for it.

Technical Foundations and Differentiators

Our platform stands on a strong technical foundation drawn from the latest research (much of it outlined in the provided documents). Key differentiators include:

- **Committor Learning for Reliability:** We adopt the concept of the *committor function* from rare-event simulation as a unifying metric for agent outcomes. In simple terms, the committor $q(x)$ measures how likely the agent, in state x , is to eventually end in a "good" outcome vs. a "bad" outcome. This idea is powerful because it's *domain-agnostic*: whether the event is "code passes all tests" vs "tests fail," or "agent completes task safely" vs "agent causes an incident," we can model it with a probabilistic score. By learning this function through active sampling (using expensive ground-truth oracle calls only for uncertain cases), we efficiently map out the boundary between success and failure. This lets us do things like **pre-emptively stop an agent when the risk of failure is too high**, or route the task to a human fallback when confidence is low. Competing solutions usually treat reliability in an ad-hoc way (or not at all), whereas we offer a principled, Bayesian approach to *quantify and reduce uncertainty* in agent performance.
- **Adaptive Red-Teaming with Epistemic Exploration:** Unlike traditional red-teaming or QA, which might run a fixed script of test cases, our system is **adaptive and ever-learning**. We employ an *Epistemic Neural Network* (ENN) or similar ensemble-based method to gauge the model's uncertainty about whether a given input will elicit a safe or unsafe outcome. The system then *generates new attack queries specifically to resolve those uncertainties*. For example, if the agent has never seen a

particular tool-use scenario and the model isn't sure if it would handle it safely, our Attacker module will create variations of that scenario to test it. This approach explicitly aims to **find unknown unknowns**. As noted, standard RL-based red teaming tends to collapse to known exploits (hammering the same jailbreak trick), whereas our approach *“maintains diversity by seeking regions of high epistemic uncertainty, ensuring the red team uncovers new modes of attack rather than exploiting known ones.”* Diversity is further ensured by optimizing for not just success rate but also *coverage of different hazard categories*, preventing mode-collapse into one style of attack. The result is a much more **comprehensive safety evaluation** that maps the true boundary of the agent's capabilities and vulnerabilities.

- **Tool-Use Safety Boundaries & Contextual Attacks:** A major gap in current tools is understanding the *sequence-based* or *context-based* failure modes unique to agentic systems. Our platform explicitly tackles these. We track not just whether an output is toxic, but whether a **sequence of actions** is dangerous. For instance, an agent might make a series of benign tool calls that in combination lead to a breach (e.g. reading a passwords file then sending its content over an API). We model such scenarios by representing the agent's state (including action history and tool context) in the committor as well. This allows the system to estimate probabilities of complex event sequences (like “Will the agent eventually perform an unauthorized action given the current trajectory?”). Using an **Oracle simulator** (e.g., a sandbox environment for the agent), we can actually let the agent run through multi-step traces to see if it falls for an indirect prompt injection or a long-horizon trap. Our adaptive loop then adjusts, *probing variations of that sequence* to see what factors truly cause the failure. This focus on **state-based vulnerabilities** – as opposed to just one-turn prompts – is a key differentiator. It gives our customers insight into how an agent might fail over time or when interacting with external systems, which is critical for tool-enabled AI. Essentially, we draw the “safety boundary” around the agent's tool use: mapping out which tool inputs and sequences stay safe (State A) versus lead to failure or harm (State B).
- **Integrated Trust and Verification Layer:** As a complement to the above, our platform can embed a **cryptographic trust layer** (inspired by solutions like Crittora). This isn't just a passive logging feature – it's an active part of safety. For example, by requiring that each instruction to the agent is signed by an authorized source, we can prevent many prompt injection attacks outright (the agent would refuse to act on any input that isn't verifiably from a trusted user or system). Likewise, by signing the agent's outputs and tool calls, we create **verifiable receipts** of everything the agent did – enabling after-the-fact auditing and forensic analysis in case of an incident. This appeals to enterprise customers who operate in regulated environments, as it provides a clear compliance trail (“Who gave the instruction?” “Did the agent deviate from policy?” – all verifiable). Competing offerings in AI safety rarely address this *operational trust* aspect; by including it, we not only help find failure modes but also help prevent and trace them in live deployments. It's a holistic approach from testing to runtime enforcement.

In summary, our platform's technical DNA – **committor-based reliability modeling**, **uncertainty-driven adaptive testing**, and **cryptographic proof of correctness** – directly stems from cutting-edge research (as evidenced in the attached documents) and goes beyond what current AI risk management products offer. It gives us a strong moat in both intellectual approach and practical effectiveness.

Potential Early Customers and Markets

We see several high-need sectors that would be early customers of this solution:

- **Developer Tooling and CI/CD Platforms:** Companies like GitHub (Microsoft), GitLab, Atlassian, or CircleCI that are adding AI features to assist in coding or code review. They could integrate our reliability engine to **filter or prioritize AI-generated code suggestions** – e.g. warn if an AI-suggested patch is very likely to fail tests, saving CI resources. This improves user trust in their AI features by reducing frivolous errors.
- **Enterprise Software Providers with AI Assistants:** Many B2B software firms (Salesforce with Einstein GPT, Oracle, ServiceNow, etc.) are embedding AI agents into their products. These agents might execute actions like updating records, sending emails, or configuring systems. Such companies are potential customers for our safety verification layer to **validate that their AI assistant won't perform unauthorized operations**. For example, an enterprise workflow tool with an AI agent could use our platform to red-team the agent against internal policies (ensuring it can't be tricked into leaking data or making forbidden changes).
- **Financial and Healthcare Institutions:** Banks, trading firms, and healthcare organizations are cautiously exploring AI agents (for customer support, automation, data analysis) but face strict regulatory and security requirements. These institutions would value a product that offers **continuous auditing and risk modeling for AI decisions**. A bank could use our system to guarantee an AI trader agent stays within risk limits (using committor models to estimate crash or loss probabilities), or a hospital could verify an AI scheduling assistant doesn't expose patient data (with our tool-use safety tests). The **"Why now"** is strong here given heightened scrutiny on AI compliance.
- **AI Model Vendors and Platforms:** Companies like OpenAI, Anthropic, or Hugging Face could be partners or customers, using our platform to **enhance their safety evaluation process**. They currently employ red-teamers and evals; our adaptive approach could make that more thorough. Even open-source AI communities (e.g. developers of Llama variants) might use an open version of our tool to certify models against standard safety tests (turning our platform into a "UL safety test" for AI). This expands our reach and could become an industry standard process for launching new agentic AI models.
- **Government and Defense Agencies:** Organizations like the Department of Defense or Homeland Security (or their counterparts globally) are investing in AI agents for logistics,

intelligence analysis, and autonomous systems. They have an acute need for verification. Our system can serve as an **AI test range** for them – actively challenging an AI agent to ensure it cannot be subverted by adversaries and that it remains within defined rules of engagement. Early contracts in this space not only provide revenue but also credibility (as these agencies often set guidelines that industry follows).

Each of these customer segments has a clear stake in **agent reliability and safety**, and budgets to invest in mitigating AI risks. We would prioritize those where the pain is most immediate (developer tools and enterprise AI assistants) for our initial go-to-market, while engaging with bigger but slower-moving clients (finance, gov) for pilots and feedback.

Prototype / MVP Suggestions

To demonstrate value quickly, we should build a targeted prototype focusing on one use-case that showcases our platform's core loop:

- **MVP 1: “SafeCoder” – AI Code Patch Validator.** This prototype targets the code-fixing agent scenario. We take an open-source code-autofix agent (or a simplified version of the SWE-bench setup) and integrate our reliability engine. The MVP would involve training a small ENN model on a dataset of code patches (successful vs failed fixes) to predict the $q(x)$ of success. We'd implement the active sampling loop: when a new patch is proposed by the agent, our system decides whether to run it in the test sandbox or skip it based on predicted success and uncertainty thresholds (using a criterion similar to BICEP). Over a week, we could demonstrate that this loop achieves significant compute savings – e.g. *“our model avoids running ~50% of the failing patches, cutting the cost per resolved issue in half”*. This MVP directly appeals to engineering teams and gives a clear numeric ROI. It's also implementable quickly with existing tools (Docker test harness, a pre-trained code embedding model for state representation, etc.).
- **MVP 2: “Auto Red Team for Tool-Using Chatbot.”** In parallel or next, we build a prototype of the adaptive safety tester using an open-source agent like a mini GPT-4 with tools. The MVP would involve a **mock agent environment** (few simulated tools like a fake file system and web search) and our attacker/judge loop. We can start with a known set of seed jailbreak prompts (from something like WildChat or AdvBench) and let our attacker module mutate them using simple strategies (e.g. rephrasing, adding contexts). We'd integrate a safety classifier (open model or even heuristic) as the Judge. Over a series of iterations, measure how the system finds diverse new failure cases that weren't in the seed set – for example, tricking the agent into making an unsafe file operation via a multi-step prompt that static tests didn't cover. Success metrics for the demo include **Attack Success Rate vs. queries** and **diversity of exploits found** (we expect to show we find significantly more diverse attack strategies than a baseline like an RL fine-tuning or random fuzzing would). This prototype will impress stakeholders

concerned with security, showing an automated way to stay ahead of evolving threats.

- **Unified Dashboard:** As part of the MVP, we can develop a simple dashboard or report format that illustrates the “*reliability map*” and “*safety map*” of an agent. For example, display a chart of the committor probability across different input conditions, highlighting a “gray zone” where the agent is 50/50 and needs more data – this helps users visually understand their model’s limits. Likewise, show clusters of discovered attack prompts on an embedding plot, with red dots for unsafe outcomes – a **visual map of the agent’s safe vs unsafe regions**. Even if rudimentary, such visualization will drive home the value of our approach in a user-friendly way.

These MVPs align tightly with the content in the provided documents, reusing their suggested frameworks and prototypes (e.g., the SWE-bench committor model and the LLM red teaming loop). By starting with these concrete implementations, we can rapidly iterate with real data and prove out the core technology. Early results from the prototypes – like cost savings percentages or novel exploits found – will be crucial for attracting pilot customers and investors. From there, we can expand the system’s coverage (more agent types, more tools, polished cryptographic integration, etc.) knowing that the foundational loop is sound.

In conclusion, this startup direction zeroes in on an **urgent, unmet need** in the AI market: making autonomous agents trustworthy and robust. By synthesizing ideas of adaptive red teaming, committor-based reliability estimation, tool-use safety boundaries, epistemic uncertainty modeling, and even cryptographic proof, our proposed solution is both novel and grounded in proven concepts. It offers a clear immediate benefit (safer and more reliable AI deployments) with strong “why now” justification and a runway to become the standard platform for AI agent assurance. The combination of immediate wedge use-cases and a broad vision gives us a compelling story for a venture – one that could start with saving a company’s compute budget or preventing a security incident, and ultimately become the **go-to guardian for agentic AI** in the coming era.

Sources: The concepts and technical frameworks discussed above are drawn and synthesized from the provided materials, including the committor-based active learning architecture, examples of its application to agent reliability and red teaming, detailed design for adaptive safety testing in tool-use agents, and emerging industry approaches to cryptographic trust in AI agents. These sources collectively illustrate the feasibility and timeliness of the proposed startup direction.

You said:

So what is that really saying then

ChatGPT said:

It’s basically saying this:

You shouldn't try to "sell CapSeal" as a grand philosophical platform first. You should sell a **very specific thing that buyers already feel pain for today**: *"I want to use AI agents, but I don't trust them, and testing them is expensive."* Your wedge is becoming the layer that **measures, predicts, and proves** whether an agent will behave correctly *before* people let it touch prod.

There are two big consensus threads across your docs.

First thread: "Agents are dangerous in a new way."

Once an LLM is allowed to call tools (files, SQL, web, code execution), the safety problem stops being "did it generate a naughty sentence?" and becomes "did it do something operationally bad?" That's what your doc calls "state-based vulnerability": the agent can output something that looks harmless while taking a destructive action through tools or being hijacked via retrieved context / prompt injection. Static benchmarks miss this because the failures often show up only in multi-step traces, tool chains, and context poisoning.

Second thread: "You already have the right math to make this a product."

Your BICEP/ENN/FusionAlpha loop is essentially an engine for mapping a boundary: where things flip from success→failure or safe→unsafe, and doing it efficiently by spending expensive "oracle" evaluations only where it's uncertain. In committor language: given a state ss , learn $q(s)=P(\text{success before failure})$. The important region is the "tube" where $q \approx 0.5$: that's where extra testing, retries, or guardrails matter most.

So the "real" startup translation is:

Build a product that gives companies a **reliability/safety risk score for agent actions**, and a **closed-loop testing system** that automatically discovers the edge cases that break their agent (especially tool-use and prompt-injection edge cases), instead of them finding out in production.

Why this is a wedge with immediate buyers:

- Buyers already pay for CI, QA, security, incident response, and compliance. "Agent misbehavior" is just those costs wearing a new hat.
- You can sell into a narrow, painful slice first: **agentic coding / AI PRs**. People are running tons of expensive evals (tests, containers, CI minutes) on agent outputs that fail most of the time. A reliability filter that prevents "obvious failures" from even running is immediate ROI.
- Parallel slice: **agentic tool-use safety** for enterprises (internal copilots that can read/write docs, query databases, trigger tickets). They need proof it won't be tricked into doing something dumb or illegal. The closed-loop adaptive red-teaming pipeline in your doc is exactly aimed at discovering those failures.

In other words: it's telling you to productize "trust" into something operational:

1. predict success/failure probability for an agent action,
2. automatically search for the boundary cases where it breaks,
3. produce artifacts (reports/logs/optionally receipts) that let a team ship with confidence.

If you want the simplest “do this next” interpretation: pick one wedge (I’d pick coding agents first), and build a minimal system that outputs: “Run this patch? YES/NO + confidence + why,” then prove you reduce wasted test runs by a large %. That turns the theory into money.


You said:

Do you see the chats in this project folder?

ChatGPT said:

Yes — I can see the **chat transcripts / conversation logs that are included in this project context** (the “Project Conversation Content” + “Recent Conversation Content” sections, with timestamps and excerpts).

What I *don’t* automatically see is whatever random files happen to be sitting in your local repo directory unless they’ve been uploaded or surfaced to me through the file tools. So:

- “Chats in the project folder” meaning the **embedded project chat history** you see above →  I see those excerpts.
- “Chats in your filesystem folder” meaning like `/home/ryan/BEF-main/...` chat logs → I only see them if you upload them or point me to them via an available file/search tool.

Net: I can reference the chat history that’s already in this project context right now, and I can search the uploaded PDFs/TXTs deeply.

You said:

Do you see the work ive been doing here so far

ChatGPT said:

Yes — I see a pretty coherent arc of work, not just random tinkering.

You’ve been doing three intertwined things:

1. Building **CapSeal/Capsule** as “proof-carrying software engineering.”
You’ve got a CLI + artifacts pipeline that can trace a project, generate receipts, verify them, audit hash chains, open rows with membership proofs, detect tampering, etc. The E2E smoke output you pasted is basically: demo → verify receipt → trace → verify

fixture → audit fixture → row opening → tamper detection. That's real infrastructure work, not vibes.

2. Trying to turn that into **agentic refactoring that's actually safe to apply**.
The product tension you kept hitting: "I don't just want a cute CLI — I need multi-agent refactoring now." You noticed a concrete gap: you had modules like `orchestrate` / `triage` / `workflow_cmd` that existed in code but weren't wired into the shell UX, meaning the product promise ("multi-agent") wasn't actually reachable from the interface yet. That's an actual product/engineering mismatch you identified and started closing.
3. In parallel, you've been pushing the **BICEP** → **ENN** → **FusionAlpha** stack as a generalized "boundary mapping" engine.
You've been testing whether you were truly running the full stack vs accidentally running stub baselines (MLP baseline, stubbed BICEP, cheap Python fusion instead of Rust). That's you catching the classic failure mode in research-y systems: *thinking* you're running the real thing while you're running the toy.

So yes: I see you doing the "hard boring spine" work (verification artifacts, receipts, auditability, E2E runs) while also trying to land the "sexy product layer" (multi-agent refactoring / orchestrated workflows), and keeping a separate research engine alive (committor / boundary learning / adaptive sampling) that could become a wedge.

One meta-observation: the work is converging toward a single theme — "agents that can act, but with enforceable trust boundaries." CapSeal is your trust substrate; the boundary-mapping loop is your way to *predict* when an agent will succeed/fail or stay safe/unsafe; the multi-agent orchestration is the UX/product expression.

If you keep pushing it, the biggest risk isn't "you haven't done enough." It's "you built 3 powerful layers and need to pick the one that prints money first," then let the other two become your moat.

ChatGPT can make mistakes. Check important info.

AI Agent Safety & Reliability Platform – Startup Proposal

Proposed Startup Idea

We propose a platform that **ensures autonomous AI agents are both reliable and safe in their tool use**. This system combines *reliability modeling* with *active safety verification* to give

organizations confidence in agents that can write code, execute tools, or manage data. The core innovation is an **adaptive verification loop** that continuously maps out an agent's "failure boundary" – the conditions under which the agent might produce errors or unsafe actions – and intervenes or informs the user before things go wrong. In practice, this means two integrated capabilities:

- **Predictive Reliability Scoring:** The platform learns a **committor function** for each agent or task – essentially the probability that the agent will succeed at a given task before failing. For example, given an agent's current state (its plan, context, or code changes pending), the system estimates the probability of a correct outcome vs. a failure. If an agent-generated code patch has a very low predicted success probability, the platform flags it or filters it out preemptively. This saves costly trial-and-error runs and prevents obvious failures from ever executing. In essence, the committor serves as a **universal reliability metric** for agent behavior in high-dimensional scenarios.
- **Active Safety "Red Teaming":** Simultaneously, the platform runs an **adaptive adversarial testing loop** against the agent to probe its safety. Rather than relying on static harmful prompt lists or one-off pen-tests, it uses an *active learning approach* to automatically discover "**semantic jailbreaks**" and **unsafe tool use sequences** that could exploit the agent. A dedicated "Attacker" module generates adversarial scenarios (malicious prompts, tool misuse sequences, etc.) and a "Judge" module evaluates if the agent fails (e.g. produces a toxic output or unauthorized action). Crucially, each attack is chosen based on *epistemic uncertainty*: the system focuses on borderline cases where it isn't sure if the agent will refuse or comply, rather than repeating known safe or known unsafe inputs. Over time this creates a topological map of the agent's vulnerability space, clustering known failure modes and **probing into nearby unknown regions of high uncertainty to find new ones**. This closed-loop process reveals complex multi-step failure modes that static benchmarks miss, such as indirect prompt injections or tool misuse that only occur in certain contexts.

Behind the scenes, these two capabilities share a common technical framework: an **Adaptive Epistemic Discovery Loop**. This loop uses an **Epistemic Neural Network (ENN)** to model what the agent *doesn't yet know* about its success/failure boundary, and a smart sampler (e.g. Bayesian criteria like BICEP) to query the "ground truth" (by running the agent in a sandbox or simulator) only on the most informative cases. By targeting areas where the predicted success probability $q(x)$ is around 0.5 and model uncertainty is high, the system efficiently learns the dividing line between safe and unsafe (or success and failure) behaviors. This makes testing far more efficient than brute-force fuzzing: rather than waste compute on millions of trivial cases (which are almost always safe) or known extreme bad cases, it hones in on the critical *edge cases*. As a result, the platform can discover high-severity failures with orders of magnitude fewer queries than random testing, **saving time and cost**.

Finally, the platform can optionally provide **cryptographic verification ("receipts")** for agent actions to enhance trust. Every tool invocation or external action by the agent can be recorded

with a cryptographic signature or proof, creating an audit trail that the action was authorized and not tampered with. This approach, inspired by emerging “zero-trust” agent frameworks, **ensures the agent only acts on inputs that are authentic and authorized**. For example, if the agent is instructed to execute a database query, the instruction can be verified as coming from a legitimate source and the query results signed. These *cryptographic receipts* pair with the reliability/safety modeling to give end-users and regulators high confidence: not only is the agent *tested* to be robust, but every runtime action is *verifiable* and auditable.

In summary, the startup would deliver a comprehensive **AI Agent Safety & Reliability Platform**. It proactively **models and bounds the risk** of deploying autonomous agents by (a) predicting and preventing likely failures, (b) systematically uncovering novel failure modes (with a focus on tool-use safety boundaries), and (c) establishing trust through verification and audit trails. This de-risks the adoption of powerful agentic AI in real-world applications.

Market Wedge and Early Adopter Profile

Our initial wedge is to target AI systems where failure is **both costly and frequent enough to be a pain point**. A prime example is the emerging class of **LLM-powered coding agents** and co-pilots. These agents propose code changes or fixes, but running their suggestions through full test suites or CI/CD pipelines is expensive (both in cloud compute and developer time). Early trials (e.g. SWE-bench for code-fixing agents) show that success rates are only ~20%, meaning 80% of test executions are wasted on bad outputs. Our platform can immediately save costs here by **filtering out low-probability “dead-on-arrival” attempts** – if the predicted $q(x)$ of success for a given code patch is near 0, the system advises skipping the full test run. This lets dev teams focus their expensive verification on the 20% of cases with promise, effectively doubling efficiency. Early adopters in this profile are likely **software companies integrating AI into their DevOps or QA processes** (e.g. a GitHub Copilot extension for auto-fixing pull requests, or enterprises using GPT-4 to refactor legacy code). They feel the pain of unreliable agent outputs as cloud bills and slow iteration cycles, so they will pay for a solution that improves throughput.

Another strong early-adopter profile is organizations deploying **LLM agents with tool access in high-stakes environments** (finance, healthcare, enterprise automation). For instance, a bank building a chatbot that can execute transactions, or a healthcare AI that can call internal databases. Such players have a *low tolerance for unsafe behavior*: a single unauthorized action or security lapse could be catastrophic. Our platform’s ability to **actively “red team” an agent’s tool-use and uncover hidden failure modes** is extremely valuable here. We move beyond standard prompt filtering to simulate complex attack scenarios an agent might face – e.g. a malicious document that poisons the agent’s context, or a sequence of benign-looking tool calls that together exfiltrate data. Early adopters in this category are **enterprises and SaaS providers adding agentic AI features** (for example, an enterprise software suite adding an “AI assistant” that can update customer records via an API). They need assurance that this assistant won’t inadvertently violate security or compliance. Our system offers them a proactive

safety net: a way to *demonstrate* and *continuously ensure* that the agent stays within safe boundaries.

In both cases, the wedge is providing **tangible risk reduction and cost savings immediately**. We are not asking customers to overhaul their AI models or pipelines – our platform slots in as a testing and guardrail layer around existing agents. This ease of integration is key for early adoption. We would offer simple APIs or a dashboard: e.g. submit an agent's action for a reliability score, or run the agent in our sandbox to get a "safety report" of potential exploits. By starting with pain points that are acute (wasted compute, security fears) and in domains where buyers have budget (developer tooling and enterprise security), we carve out a viable entry market.

Why Now

1. Agents are becoming practical and widespread: Within the last 1-2 years, large language models have evolved from just generating text to being **"agentic" – perceiving, deciding, and acting through tools**. OpenAI, Anthropic, and others are deploying API functions, plugins, and autonomous agent frameworks that let AI systems execute code, make web requests, or control real-world systems. This is a *sea change* in capability that also introduces new risks. As noted in our research, *"the emergence of 'Agentic AI' ... introduced a new and far more dangerous category of risk: State-Based Vulnerability."* Unlike a chatbot that could only produce a toxic sentence, an agent can actually perform harmful operations (even if its output text looks benign). The industry is only beginning to grapple with this. **The timing is perfect for a solution addressing these agent-specific risks**, as more companies move from experimental demos to real deployments of tool-using AI. Everyone is asking how to trust these agents in production.

2. Limitations of current safety approaches: The status quo for AI safety and verification is lagging behind this new paradigm. Most existing tools focus on content filtering or static red-teaming with known prompts. These are insufficient for agents that operate in **dynamic, context-rich environments**. For example, companies rely on benchmarks like JailbreakBench or a set of "forbidden prompt" tests. But as our analysis showed, *static benchmarks lack contextual awareness* (they don't cover multi-turn or tool-specific scenarios) and models quickly overfit to them, leaving blind spots. Moreover, reliability is largely unaddressed – developers currently find out an agent is unreliable only after it fails repeatedly in practice. **Recent advancements in research (including the attached frameworks) provide a new way:** using active learning and uncertainty quantification to *systematically probe an AI's weaknesses*. Techniques like the committor-based active sampling have matured in research contexts just as the need for them becomes dire in industry. This convergence of a pressing problem and an available technical solution makes now an ideal moment to build a product.

3. Technology tailwinds: Key components of our platform are now feasible where they weren't a few years ago. High-quality embedding models and vector databases (FAISS, etc.) allow us to map large semantic spaces of prompts or states efficiently – enabling the kind of graph-based

uncertainty modeling we use. Open-source safety evaluators (like Llama Guard 3 mentioned in our docs) are emerging, meaning we can build a “Judge” without needing a proprietary service. There’s also a cultural shift towards *verification and audit in AI*: initiatives in “model cards,” “AI audit trails,” and even cryptographic verification of AI outputs (as seen with companies like Crittora adding trust layers) show that the market is ready to value transparency and proof in AI interactions. All these factors lower the barrier to entry for our solution and increase the appetite for it.

Technical Foundations and Differentiators

Our platform stands on a strong technical foundation drawn from the latest research (much of it outlined in the provided documents). Key differentiators include:

- **Committer Learning for Reliability:** We adopt the concept of the *committer function* from rare-event simulation as a unifying metric for agent outcomes. In simple terms, the committer $q(x)$ measures how likely the agent, in state x , is to eventually end in a “good” outcome vs. a “bad” outcome. This idea is powerful because it’s *domain-agnostic*: whether the event is “code passes all tests” vs “tests fail,” or “agent completes task safely” vs “agent causes an incident,” we can model it with a probabilistic score. By learning this function through active sampling (using expensive ground-truth oracle calls only for uncertain cases), we efficiently map out the boundary between success and failure. This lets us do things like **pre-emptively stop an agent when the risk of failure is too high**, or route the task to a human fallback when confidence is low. Competing solutions usually treat reliability in an ad-hoc way (or not at all), whereas we offer a principled, Bayesian approach to *quantify and reduce uncertainty* in agent performance.
- **Adaptive Red-Teaming with Epistemic Exploration:** Unlike traditional red-teaming or QA, which might run a fixed script of test cases, our system is **adaptive and ever-learning**. We employ an *Epistemic Neural Network* (ENN) or similar ensemble-based method to gauge the model’s uncertainty about whether a given input will elicit a safe or unsafe outcome. The system then *generates new attack queries specifically to resolve those uncertainties*. For example, if the agent has never seen a particular tool-use scenario and the model isn’t sure if it would handle it safely, our Attacker module will create variations of that scenario to test it. This approach explicitly aims to **find unknown unknowns**. As noted, standard RL-based red teaming tends to collapse to known exploits (hammering the same jailbreak trick), whereas our approach *“maintains diversity by seeking regions of high epistemic uncertainty, ensuring the red team uncovers new modes of attack rather than exploiting known ones.”* Diversity is further ensured by optimizing for not just success rate but also *coverage of different hazard categories*, preventing mode-collapse into one style of attack. The result is a much more **comprehensive safety evaluation** that maps the true boundary of the agent’s capabilities and vulnerabilities.

- Tool-Use Safety Boundaries & Contextual Attacks:** A major gap in current tools is understanding the *sequence-based* or *context-based* failure modes unique to agentic systems. Our platform explicitly tackles these. We track not just whether an output is toxic, but whether a **sequence of actions** is dangerous. For instance, an agent might make a series of benign tool calls that in combination lead to a breach (e.g. reading a passwords file then sending its content over an API). We model such scenarios by representing the agent's state (including action history and tool context) in the committor as well. This allows the system to estimate probabilities of complex event sequences (like "Will the agent eventually perform an unauthorized action given the current trajectory?"). Using an **Oracle simulator** (e.g., a sandbox environment for the agent), we can actually let the agent run through multi-step traces to see if it falls for an indirect prompt injection or a long-horizon trap. Our adaptive loop then adjusts, *probing variations of that sequence* to see what factors truly cause the failure. This focus on **state-based vulnerabilities** – as opposed to just one-turn prompts – is a key differentiator. It gives our customers insight into how an agent might fail over time or when interacting with external systems, which is critical for tool-enabled AI. Essentially, we draw the "safety boundary" around the agent's tool use: mapping out which tool inputs and sequences stay safe (State A) versus lead to failure or harm (State B).
- Integrated Trust and Verification Layer:** As a complement to the above, our platform can embed a **cryptographic trust layer** (inspired by solutions like Crittora). This isn't just a passive logging feature – it's an active part of safety. For example, by requiring that each instruction to the agent is signed by an authorized source, we can prevent many prompt injection attacks outright (the agent would refuse to act on any input that isn't verifiably from a trusted user or system). Likewise, by signing the agent's outputs and tool calls, we create **verifiable receipts** of everything the agent did – enabling after-the-fact auditing and forensic analysis in case of an incident. This appeals to enterprise customers who operate in regulated environments, as it provides a clear compliance trail ("Who gave the instruction?" "Did the agent deviate from policy?" – all verifiable). Competing offerings in AI safety rarely address this *operational trust* aspect; by including it, we not only help find failure modes but also help prevent and trace them in live deployments. It's a holistic approach from testing to runtime enforcement.

In summary, our platform's technical DNA – **committor-based reliability modeling**, **uncertainty-driven adaptive testing**, and **cryptographic proof of correctness** – directly stems from cutting-edge research (as evidenced in the attached documents) and goes beyond what current AI risk management products offer. It gives us a strong moat in both intellectual approach and practical effectiveness.

Potential Early Customers and Markets

We see several high-need sectors that would be early customers of this solution:

- **Developer Tooling and CI/CD Platforms:** Companies like GitHub (Microsoft), GitLab, Atlassian, or CircleCI that are adding AI features to assist in coding or code review. They could integrate our reliability engine to **filter or prioritize AI-generated code suggestions** – e.g. warn if an AI-suggested patch is very likely to fail tests, saving CI resources. This improves user trust in their AI features by reducing frivolous errors.
- **Enterprise Software Providers with AI Assistants:** Many B2B software firms (Salesforce with Einstein GPT, Oracle, ServiceNow, etc.) are embedding AI agents into their products. These agents might execute actions like updating records, sending emails, or configuring systems. Such companies are potential customers for our safety verification layer to **validate that their AI assistant won't perform unauthorized operations**. For example, an enterprise workflow tool with an AI agent could use our platform to red-team the agent against internal policies (ensuring it can't be tricked into leaking data or making forbidden changes).
- **Financial and Healthcare Institutions:** Banks, trading firms, and healthcare organizations are cautiously exploring AI agents (for customer support, automation, data analysis) but face strict regulatory and security requirements. These institutions would value a product that offers **continuous auditing and risk modeling for AI decisions**. A bank could use our system to guarantee an AI trader agent stays within risk limits (using committor models to estimate crash or loss probabilities), or a hospital could verify an AI scheduling assistant doesn't expose patient data (with our tool-use safety tests). The **"Why now"** is strong here given heightened scrutiny on AI compliance.
- **AI Model Vendors and Platforms:** Companies like OpenAI, Anthropic, or Hugging Face could be partners or customers, using our platform to **enhance their safety evaluation process**. They currently employ red-teamers and evals; our adaptive approach could make that more thorough. Even open-source AI communities (e.g. developers of Llama variants) might use an open version of our tool to certify models against standard safety tests (turning our platform into a "UL safety test" for AI). This expands our reach and could become an industry standard process for launching new agentic AI models.
- **Government and Defense Agencies:** Organizations like the Department of Defense or Homeland Security (or their counterparts globally) are investing in AI agents for logistics, intelligence analysis, and autonomous systems. They have an acute need for verification. Our system can serve as an **AI test range** for them – actively challenging an AI agent to ensure it cannot be subverted by adversaries and that it remains within defined rules of engagement. Early contracts in this space not only provide revenue but also credibility (as these agencies often set guidelines that industry follows).

Each of these customer segments has a clear stake in **agent reliability and safety**, and budgets to invest in mitigating AI risks. We would prioritize those where the pain is most

immediate (developer tools and enterprise AI assistants) for our initial go-to-market, while engaging with bigger but slower-moving clients (finance, gov) for pilots and feedback.

Prototype / MVP Suggestions

To demonstrate value quickly, we should build a targeted prototype focusing on one use-case that showcases our platform's core loop:

- **MVP 1: “SafeCoder” – AI Code Patch Validator.** This prototype targets the code-fixing agent scenario. We take an open-source code-autofix agent (or a simplified version of the SWE-bench setup) and integrate our reliability engine. The MVP would involve training a small ENN model on a dataset of code patches (successful vs failed fixes) to predict the $q(x)$ of success. We'd implement the active sampling loop: when a new patch is proposed by the agent, our system decides whether to run it in the test sandbox or skip it based on predicted success and uncertainty thresholds (using a criterion similar to BICEP). Over a week, we could demonstrate that this loop achieves significant compute savings – e.g. *“our model avoids running ~50% of the failing patches, cutting the cost per resolved issue in half”*. This MVP directly appeals to engineering teams and gives a clear numeric ROI. It's also implementable quickly with existing tools (Docker test harness, a pre-trained code embedding model for state representation, etc.).
- **MVP 2: “Auto Red Team for Tool-Using Chatbot.”** In parallel or next, we build a prototype of the adaptive safety tester using an open-source agent like a mini GPT-4 with tools. The MVP would involve a **mock agent environment** (few simulated tools like a fake file system and web search) and our attacker/judge loop. We can start with a known set of seed jailbreak prompts (from something like WildChat or AdvBench) and let our attacker module mutate them using simple strategies (e.g. rephrasing, adding contexts). We'd integrate a safety classifier (open model or even heuristic) as the Judge. Over a series of iterations, measure how the system finds diverse new failure cases that weren't in the seed set – for example, tricking the agent into making an unsafe file operation via a multi-step prompt that static tests didn't cover. Success metrics for the demo include **Attack Success Rate vs. queries** and **diversity of exploits found** (we expect to show we find significantly more diverse attack strategies than a baseline like an RL fine-tuning or random fuzzing would). This prototype will impress stakeholders concerned with security, showing an automated way to stay ahead of evolving threats.
- **Unified Dashboard:** As part of the MVP, we can develop a simple dashboard or report format that illustrates the *“reliability map”* and *“safety map”* of an agent. For example, display a chart of the committor probability across different input conditions, highlighting a “gray zone” where the agent is 50/50 and needs more data – this helps users visually understand their model's limits. Likewise, show clusters of discovered attack prompts on an embedding plot, with red dots for unsafe outcomes – a **visual map of the agent's safe vs unsafe regions**. Even if rudimentary, such visualization will drive home the

value of our approach in a user-friendly way.

These MVPs align tightly with the content in the provided documents, reusing their suggested frameworks and prototypes (e.g., the SWE-bench committor model and the LLM red teaming loop). By starting with these concrete implementations, we can rapidly iterate with real data and prove out the core technology. Early results from the prototypes – like cost savings percentages or novel exploits found – will be crucial for attracting pilot customers and investors. From there, we can expand the system’s coverage (more agent types, more tools, polished cryptographic integration, etc.) knowing that the foundational loop is sound.

In conclusion, this startup direction zeroes in on an **urgent, unmet need** in the AI market: making autonomous agents trustworthy and robust. By synthesizing ideas of adaptive red teaming, committor-based reliability estimation, tool-use safety boundaries, epistemic uncertainty modeling, and even cryptographic proof, our proposed solution is both novel and grounded in proven concepts. It offers a clear immediate benefit (safer and more reliable AI deployments) with strong “why now” justification and a runway to become the standard platform for AI agent assurance. The combination of immediate wedge use-cases and a broad vision gives us a compelling story for a venture – one that could start with saving a company’s compute budget or preventing a security incident, and ultimately become the **go-to guardian for agentic AI** in the coming era.

Sources: The concepts and technical frameworks discussed above are drawn and synthesized from the provided materials, including the committor-based active learning architecture, examples of its application to agent reliability and red teaming, detailed design for adaptive safety testing in tool-use agents, and emerging industry approaches to cryptographic trust in AI agents. These sources collectively illustrate the feasibility and timeliness of the proposed startup direction.

Cross-round state, deltas, and reporting consistency

update-name-jgg

+1355-25

```
cat > /home/ryan/otherstuff/agent_bench/runner.py << 'ENDOFFILE'

#!/usr/bin/env python3

"""AgentEvalBench Runner - Round orchestration, acquisition, and posteriors."""

from __future__ import annotations

import hashlib
```

```

import json

import uuid

from datetime import datetime

from pathlib import Path

from typing import Any, Dict, List, Optional, Tuple


import numpy as np


from agent_bench.env_toy_v1 import ToyToolEnv

from agent_bench.agent_toy_v1 import ToyAgent

from agent_bench.grid import load_grid, get_params_for_idx, GRID_VERSION

from agent_bench.metrics import compute_agent_tube_metrics


BUDGET_PER_TARGET = 256


def derive_episode_seed(run_uuid: str, round_num: int, grid_idx: int, episode_idx:
int) -> int:

    """Deterministic seed derivation - stable across resume/machines/Python
versions."""

    data = f"{run_uuid}:{round_num}:{grid_idx}:{episode_idx}".encode()

    h = hashlib.blake2b(data, digest_size=8)

    return int.from_bytes(h.digest(), 'little')


def derive_agent_seed(episode_seed: int) -> int:

    """Derive separate RNG stream for agent using blake2b with different
personalization."""

```

```

    data = f"agent:{episode_seed}".encode()

    h = hashlib.blake2b(data, digest_size=8,
person=b'AGENTSEED\x00\x00\x00\x00\x00\x00\x00')

    return int.from_bytes(h.digest(), 'little')

def compute_acquisition_score(

    alpha: np.ndarray,

    beta: np.ndarray,

    tau: float = 0.2,

    sigma: float = 0.05,

    w1: float = 1.0,

    w2: float = 0.5,

) -> np.ndarray:

    """Compute acquisition scores for all grid points."""

    alpha_f = alpha.astype(np.float64)

    beta_f = beta.astype(np.float64)

    total = alpha_f + beta_f

    mu = alpha_f / total

    var = (alpha_f * beta_f) / (total ** 2 * (total + 1))

    boundary_bonus = np.exp(-np.abs(mu - tau) / sigma)

    return var * w1 + boundary_bonus * w2

def select_targets(scores: np.ndarray, K: int) -> np.ndarray:

    """Select top-K targets with deterministic tie-breaking."""

    n = len(scores)

```

```

    order = np.lexsort((np.arange(n), -scores))

    return order[:min(K, n)]

def synthetic_p_fail(
    tool_noise: int,

    verify_flip: float,

    hint_ambiguity: int,

    distractor_count: int,

    memory_tokens: int,
) -> float:

    """Closed-form failure probability surrogate for synthetic mode."""

    a, b, c, d, e = 0.9, 0.3, 0.3, 0.2, 0.2

    return float(np.clip(

        a * verify_flip +

        b * (tool_noise / 3) +

        c * (hint_ambiguity / 3) +

        d * (distractor_count / 6) +

        e * (16 / memory_tokens),

        0.0, 1.0

    ))

def run_episode(
    grid: Dict[str, np.ndarray],

    grid_idx: int,

    episode_seed: int,

```

```

    use_synthetic: bool = False,
) -> bool:

    """Run a single episode and return success/failure."""

    params = get_params_for_idx(grid, grid_idx)

    if use_synthetic:

        p_fail = synthetic_p_fail(**params)

        rng = np.random.default_rng(episode_seed)

        return rng.random() >= p_fail

    else:

        env_rng = np.random.default_rng(episode_seed)

        agent_rng = np.random.default_rng(derive_agent_seed(episode_seed))

        env = ToyToolEnv(

            tool_noise=params["tool_noise"],

            verify_flip=params["verify_flip"],

            hint_ambiguity=params["hint_ambiguity"],

            distractor_count=params["distractor_count"],

            memory_tokens=params["memory_tokens"],

            rng=env_rng,

        )

        agent = ToyAgent()

        guess = agent.act(env, agent_rng)

        return env.check_answer(guess)

```

```

def load_posteriors(run_dir: Path) -> Tuple[np.ndarray, np.ndarray, Dict[str, Any]]:

    """Load beta posteriors from run directory."""

    beta_path = run_dir / "beta_posteriors.npz"

    data = np.load(beta_path, allow_pickle=True)

    alpha = data["alpha"]

    beta = data["beta"]

    metadata = {}

    for key in ["grid_version", "run_uuid", "posterior_semantics"]:

        if key in data.files:

            metadata[key] = str(data[key])

    return alpha, beta, metadata


def save_posteriors(run_dir: Path, alpha: np.ndarray, beta: np.ndarray, run_uuid: str)
-> None:

    """Save beta posteriors atomically."""

    from loop_io import save_npz_atomic

    save_npz_atomic(

        run_dir / "beta_posteriors.npz",

        alpha=alpha.astype(np.int64),

        beta=beta.astype(np.int64),

        grid_version=np.array(GRID_VERSION),

        run_uuid=np.array(run_uuid),

        posterior_semantics=np.array("Beta over p_fail; alpha=fails+1,
beta=successes+1"),

    )

```



```

def update_posteriors(
    alpha: np.ndarray,
    beta: np.ndarray,
    results: List[Dict[str, Any]],
) -> Tuple[np.ndarray, np.ndarray]:
    """Update posteriors based on episode results."""
    alpha = alpha.copy()
    beta = beta.copy()
    for r in results:
        idx = r["grid_idx"]
        success = r["success"]
        fail = 1 - success
        alpha[idx] += fail
        beta[idx] += success
    return alpha, beta


def assert_round_artifacts_exist(round_dir: Path, expected_episodes: int) -> None:
    """Runtime assertion before committing round."""
    results_path = round_dir / "agent_results.csv"
    assert results_path.exists(), f"Missing agent_results.csv in {round_dir}"
    with open(results_path) as f:
        rows = sum(1 for _ in f) - 1
    assert rows == expected_episodes, f"Expected {expected_episodes} episodes, got {rows}"
    beta_path = round_dir.parent.parent / "beta_posteriors.npz"
    assert beta_path.exists(), f"Missing beta_posteriors.npz"

```

```
ENDOFFILE
```

```
echo "First part written"
```

```
(eval):1: permission denied: /home/ryan/otherstuff/agent_bench/runner.py
```

```
cat > /home/ryan/otherstuff/agent_bench/runner.py << 'ENDOFFILE'
```

```
#!/usr/bin/env python3
```

```
"""AgentEvalBench Runner - Round orchestration, acquisition, and posteriors."""
```

```
from __future__ import annotations
```

```
import hashlib
```

```
import json
```

```
import uuid
```

```
from datetime import datetime
```

```
from pathlib import Path
```

```
from typing import Any, Dict, List, Optional, Tuple
```

```
import numpy as np
```

```
from agent_bench.env_toy_v1 import ToyToolEnv
```

```
from agent_bench.agent_toy_v1 import ToyAgent
```

```
from agent_bench.grid import load_grid, get_params_for_idx, GRID_VERSION
```

```
from agent_bench.metrics import compute_agent_tube_metrics
```

```
BUDGET_PER_TARGET = 256
```

```
def derive_episode_seed(run_uuid: str, round_num: int, grid_idx: int, episode_idx:
int) -> int:
```

```
    """Deterministic seed derivation - stable across resume/machines/Python
versions."""
```

```
    data = f"{run_uuid}:{round_num}:{grid_idx}:{episode_idx}".encode()
```

```
    h = hashlib.blake2b(data, digest_size=8)
```

```
    return int.from_bytes(h.digest(), 'little')
```

```
def derive_agent_seed(episode_seed: int) -> int:
```

```
    """Derive separate RNG stream for agent using blake2b with different
personalization."""
```

```
    data = f"agent:{episode_seed}".encode()
```

```
    h = hashlib.blake2b(data, digest_size=8,
person=b'AGENTSEED\x00\x00\x00\x00\x00\x00\x00')
```

```
    return int.from_bytes(h.digest(), 'little')
```

```
def compute_acquisition_score(
```

```
    alpha: np.ndarray,
```

```
    beta: np.ndarray,
```

```
    tau: float = 0.2,
```

```
    sigma: float = 0.05,
```

```
    w1: float = 1.0,
```

```
    w2: float = 0.5,
```

```
) -> np.ndarray:
```

```
    """Compute acquisition scores for all grid points."""
```

```
    alpha_f = alpha.astype(np.float64)
```

```

beta_f = beta.astype(np.float64)

total = alpha_f + beta_f

mu = alpha_f / total

var = (alpha_f * beta_f) / (total ** 2 * (total + 1))

boundary_bonus = np.exp(-np.abs(mu - tau) / sigma)

return var * w1 + boundary_bonus * w2

```

```

def select_targets(scores: np.ndarray, K: int) -> np.ndarray:

    """Select top-K targets with deterministic tie-breaking."""

    n = len(scores)

    order = np.lexsort((np.arange(n), -scores))

    return order[:min(K, n)]

```

```

def synthetic_p_fail(
    tool_noise: int,
    verify_flip: float,
    hint_ambiguity: int,
    distractor_count: int,
    memory_tokens: int,
) -> float:

    """Closed-form failure probability surrogate for synthetic mode."""

    a, b, c, d, e = 0.9, 0.3, 0.3, 0.2, 0.2

    return float(np.clip(
        a * verify_flip +
        b * (tool_noise / 3) +

```

```

        c * (hint_ambiguity / 3) +

        d * (distractor_count / 6) +

        e * (16 / memory_tokens),

        0.0, 1.0

    ))

def run_episode(

    grid: Dict[str, np.ndarray],

    grid_idx: int,

    episode_seed: int,

    use_synthetic: bool = False,
) -> bool:

    """Run a single episode and return success/failure."""

    params = get_params_for_idx(grid, grid_idx)

    if use_synthetic:

        p_fail = synthetic_p_fail(**params)

        rng = np.random.default_rng(episode_seed)

        return rng.random() >= p_fail

    else:

        env_rng = np.random.default_rng(episode_seed)

        agent_rng = np.random.default_rng(derive_agent_seed(episode_seed))

        env = ToyToolEnv(

            tool_noise=params["tool_noise"],

            verify_flip=params["verify_flip"],

```

```

        hint_ambiguity=params["hint_ambiguity"],

        distractor_count=params["distractor_count"],

        memory_tokens=params["memory_tokens"],

        rng=env_rng,

    )

```

```

    agent = ToyAgent()

    guess = agent.act(env, agent_rng)

    return env.check_answer(guess)

```

```

def load_posteriors(run_dir: Path) -> Tuple[np.ndarray, np.ndarray, Dict[str, Any]]:

```

```

    """Load beta posteriors from run directory."""

```

```

    beta_path = run_dir / "beta_posteriors.npz"

```

```

    data = np.load(beta_path, allow_pickle=True)

```

```

    alpha = data["alpha"]

```

```

    beta = data["beta"]

```

```

    metadata = {}

```

```

    for key in ["grid_version", "run_uuid", "posterior_semantics"]:

```

```

        if key in data.files:

```

```

            metadata[key] = str(data[key])

```

```

    return alpha, beta, metadata

```

```

def save_posteriors(run_dir: Path, alpha: np.ndarray, beta: np.ndarray, run_uuid: str)
-> None:

```

```

    """Save beta posteriors atomically."""

```

```

from loop_io import save_npz_atomic

save_npz_atomic(

    run_dir / "beta_posteriors.npz",

    alpha=alpha.astype(np.int64),

    beta=beta.astype(np.int64),

    grid_version=np.array(GRID_VERSION),

    run_uuid=np.array(run_uuid),

    posterior_semantics=np.array("Beta over p_fail; alpha=fails+1,
beta=successes+1"),

)

def update_posteriors(

    alpha: np.ndarray,

    beta: np.ndarray,

    results: List[Dict[str, Any]],

) -> Tuple[np.ndarray, np.ndarray]:

    """Update posteriors based on episode results."""

    alpha = alpha.copy()

    beta = beta.copy()

    for r in results:

        idx = r["grid_idx"]

        success = r["success"]

        fail = 1 - success

        alpha[idx] += fail

        beta[idx] += success

    return alpha, beta

```

```

def assert_round_artifacts_exist(round_dir: Path, expected_episodes: int) -> None:

    """Runtime assertion before committing round."""

    results_path = round_dir / "agent_results.csv"

    assert results_path.exists(), f"Missing agent_results.csv in {round_dir}"

    with open(results_path) as f:

        rows = sum(1 for _ in f) - 1

        assert rows == expected_episodes, f"Expected {expected_episodes} episodes, got {rows}"

        beta_path = round_dir.parent.parent / "beta_posteriors.npz"

        assert beta_path.exists(), f"Missing beta_posteriors.npz"

ENDOFFILE

echo "First part written"

```

First part written

```

cat >> /home/ryan/otherstuff/agent_bench/runner.py << 'ENDOFFILE'

def _metrics_dict_to_round_metrics(metrics: Dict[str, Any]) -> Any:

    """Convert metrics dict to RoundMetrics-compatible object for summary CSV."""

    from loop_metrics import (

        TubeMetrics, PlanMetrics, ModelChangeMetrics,

        CountMetrics, FusionMetrics, RoundMetrics

    )

    tube = metrics["tube"]

    counts = metrics["counts"]

```



```

plan = metrics.get("plan", {})

tube_metrics = TubeMetrics(

    tube_var_sum=tube["tube_var_sum"],

    tube_var_delta=tube.get("tube_var_delta_prev"),

    tube_var_delta_prev=tube.get("tube_var_delta_prev"),

    tube_var_delta_baseline=tube.get("tube_var_delta_baseline"),

    tube_var_baseline=tube.get("tube_var_baseline"),

    tube_coverage=tube["tube_coverage"],

    tube_points_total=tube["tube_points_total"],

    beta_std_mean_tube=tube.get("epistemic", {}).get("mean", 0.0),

    epistemic=tube.get("epistemic", {"mean": 0.0, "max": 0.0, "median": 0.0}),

    aleatoric=tube.get("aleatoric", {"mean": 0.0, "max": 0.0, "median": 0.0}),

)

plan_metrics = PlanMetrics(

    targets_selected=plan.get("targets_selected", 0),

    total_budget=plan.get("total_budget", 0),

    top_targets=[],

)

model_change = ModelChangeMetrics(enn_changed=False, fusion_changed=False)

count_metrics = CountMetrics(

    sampled_points_new=counts["sampled_points_new"],

    sampled_points_total=counts["sampled_points_total"],

    trajectories_new=counts.get("trajectories_new", 0),

    trajectories_total=counts.get("trajectories_total", 0),

```

```

    )

    fusion_metrics = FusionMetrics(status="OK", mu_offset=0.0)

    return RoundMetrics(
        round_id=metrics["round_id"],
        timestamp=datetime.now().isoformat(),
        seed=0,
        status=metrics["status"],
        tube=tube_metrics,
        plan=plan_metrics,
        model_change=model_change,
        counts=count_metrics,
        fusion=fusion_metrics,
    )

def run_one_agent_round(
    run_dir: Path,
    round_num: int,
    run_uuid: str,
    grid: Dict[str, np.ndarray],
    targets_per_round: int,
    episodes_per_budget_unit: int,
    use_synthetic: bool,
    verbose: bool,
    prev_tube_var: Optional[float],
    baseline_tube_var: Optional[float],

```

```

) -> Tuple[Dict[str, Any], bool]:

    """Run one complete round of agent evaluation."""

    from loop_io import save_json, save_csv_atomic, append_to_summary_csv

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

    round_id = f"R{round_num:04d}_{timestamp}"

    print(f"\n{' '*70}")

    print(f" AGENT ROUND {round_num} ({round_id}) ".center(70))

    print(f"{' '*70}\n")

    round_dir = run_dir / "rounds" / round_id

    round_dir.mkdir(parents=True, exist_ok=True)

    # Save round_pre.json

    round_pre = {

        "round_id": round_id,

        "round_num": round_num,

        "run_uuid": run_uuid,

        "timestamp": timestamp,

        "mode": "agent_eval",

        "use_synthetic": use_synthetic,

    }

    save_json(round_pre, round_dir / "round_pre.json")

    # Load posteriors and compute acquisition

    alpha, beta, _ = load_posteriors(run_dir)

```

```

scores = compute_acquisition_score(alpha, beta)

K = min(targets_per_round, len(scores))

selected = select_targets(scores, K)

episodes_per_target = BUDGET_PER_TARGET * episodes_per_budget_unit

total_episodes = len(selected) * episodes_per_target

print(f"[Round] Selected {len(selected)} targets, {episodes_per_target} episodes
each")

print(f"[Round] Total episodes: {total_episodes}")

print(f"[Round] Mode: {'Synthetic' if use_synthetic else 'Real simulation'}")

# Save plan
plan = {
    "round_id": round_id,
    "selected": selected.tolist(),
    "budget_per_target": BUDGET_PER_TARGET,
    "episodes_per_budget_unit": episodes_per_budget_unit,
    "total_episodes": episodes_per_target,
    "K": K,
    "tau": 0.2,
    "sigma": 0.05,
    "w1": 1.0,
    "w2": 0.5,
    "seed": None,
}

save_json(plan, round_dir / "active_sampling_plan.json")

```

```

# Run episodes

results: List[Dict[str, Any]] = []

for target_idx, grid_idx in enumerate(selected):

    grid_idx = int(grid_idx)

    params = get_params_for_idx(grid, grid_idx)

    for episode_idx in range(epochs_per_target):

        episode_seed = derive_episode_seed(run_uuid, round_num, grid_idx,
episode_idx)

        success = run_episode(grid, grid_idx, episode_seed, use_synthetic)

        results.append({

            "round_id": round_id,

            "grid_idx": grid_idx,

            "episode_idx": episode_idx,

            "episode_seed": episode_seed,

            "success": 1 if success else 0,

            **params,

        })

    if verbose and (target_idx + 1) % 10 == 0:

        print(f"  Completed {target_idx + 1}/{len(selected)} targets")

# Save agent_results.csv (atomic)

columns = [

```

```

        "round_id", "grid_idx", "episode_idx", "episode_seed", "success",

        "tool_noise", "verify_flip", "hint_ambiguity", "distractor_count",
"memory_tokens",

    ]

    save_csv_atomic(round_dir / "agent_results.csv", results, columns)

    # Update posteriors

    alpha_new, beta_new = update_posteriors(alpha, beta, results)

    save_posteriors(run_dir, alpha_new, beta_new, run_uuid)

    # Compute metrics

    metrics = compute_agent_tube_metrics(

        alpha=alpha_new,

        beta=beta_new,

        round_id=round_id,

        round_num=round_num,

        prev_tube_var=prev_tube_var,

        baseline_tube_var=baseline_tube_var,

        selected=selected,

        episodes_per_target=episodes_per_target,

    )

    # Assertion before commit

    assert_round_artifacts_exist(round_dir, total_episodes)

    # Save metrics.json (COMMIT MARKER)

    save_json(metrics, round_dir / "metrics.json")

```

```
# Append to summary.csv

metrics_obj = _metrics_dict_to_round_metrics(metrics)

append_to_summary_csv(run_dir, metrics_obj, run_uuid=run_uuid)


# Save round_post.json

round_post = {

    "round_id": round_id,

    "completed_at": datetime.now().isoformat(),

    "status": metrics["status"],

    "tube_var_sum": metrics["tube"]["tube_var_sum"],

}

save_json(round_post, round_dir / "round_post.json")


print(f"\n[Round {round_num}] Complete")

print(f"  tube_var_sum: {metrics['tube']['tube_var_sum']:.6f}")

print(f"  tube_coverage: {metrics['tube']['tube_coverage']:.3f}")

print(f"  status: {metrics['status']}")


return metrics, True

ENDOFFILE

echo "Second part written"
```

Second part written

```
cat >> /home/ryan/otherstuff/agent_bench/runner.py << 'ENDOFFILE'
```

```

def run_agent_eval_loop(
    run_dir: Path,
    n_rounds: int,
    base_seed: int,
    seed_mode: str,
    agent_bench: str = "toy_v1",
    episodes_per_budget_unit: int = 1,
    targets_per_round: int = 64,
    use_synthetic: bool = False,
    verbose: bool = False,
) -> None:
    """Run the full agent evaluation loop."""

    run_dir = Path(run_dir)

    if agent_bench != "toy_v1":
        raise ValueError(f"Unknown agent benchmark: {agent_bench}. Only 'toy_v1' is supported.")

    # Load or create run metadata
    metadata_path = run_dir / "run_metadata.json"

    if metadata_path.exists():
        with open(metadata_path) as f:
            metadata = json.load(f)

            run_uuid = metadata.get("run_uuid", str(uuid.uuid4())[:8])

            print(f"[Init] Resuming run: {run_uuid}")
    else:
        run_uuid = str(uuid.uuid4())[:8]

```



```

metadata = {

    "run_uuid": run_uuid,

    "base_seed": base_seed,

    "seed_mode": seed_mode,

    "mode": "agent_eval",

    "agent_bench": agent_bench,

    "created_at": datetime.now().isoformat(),

}

with open(metadata_path, "w") as f:

    json.dump(metadata, f, indent=2)

print(f"[Init] New run created: {run_uuid}")


# Load grid

grid_path = run_dir / "grid.npz"

if not grid_path.exists():

    raise FileNotFoundError(

        f"Grid not found at {grid_path}. "

        f"Generate one with: python -m agent_bench.grid --out {grid_path}"

    )

grid = load_grid(grid_path)

n_points = int(grid["n_points"])

print(f"[Init] Loaded grid with {n_points} points")


# Initialize beta_posteriors if missing

beta_path = run_dir / "beta_posteriors.npz"

if not beta_path.exists():

    alpha = np.ones(n_points, dtype=np.int64)

```

```

        beta = np.ones(n_points, dtype=np.int64)

        save_posteriors(run_dir, alpha, beta, run_uuid)

        print(f"[Init] Created beta_posteriors.npz with {n_points} points (Beta(1,1)
prior)")

    (run_dir / "rounds").mkdir(exist_ok=True)

    # Determine starting round

    from loop_io import get_next_round_num

    start_round = get_next_round_num(run_dir)

    end_round = start_round + n_rounds - 1

    # Load baseline from previous rounds if resuming

    baseline_tube_var: Optional[float] = None

    prev_tube_var: Optional[float] = None

    if start_round > 1:

        from loop_io import get_previous_round_info

        prev_info = get_previous_round_info(run_dir, current_round_num=start_round)

        if prev_info:

            baseline_tube_var = prev_info.get("baseline_tube_var")

            prev_tube_var = prev_info.get("tube_var")

            print(f"[Resume] Continuing from round {start_round}")

            if baseline_tube_var:

                print(f"[Resume] Loaded baseline tube_var: {baseline_tube_var:.6f}")

    print(f"""

```

```

{'='*70}

{'AGENT EVALUATION LOOP'.center(70)}

{'='*70}

Run UUID:          {run_uuid}

Rounds to run:      {n_rounds} (R{start_round:04d} -> R{end_round:04d})

Grid points:        {n_points}

Targets/round:      {targets_per_round}

Episodes/target:    {BUDGET_PER_TARGET * episodes_per_budget_unit}

Mode:               {'Synthetic' if use_synthetic else 'Real simulation'}

{'='*70}

"""

    for round_num in range(start_round, end_round + 1):

        metrics, success = run_one_agent_round(

            run_dir=run_dir,

            round_num=round_num,

            run_uuid=run_uuid,

            grid=grid,

            targets_per_round=targets_per_round,

            episodes_per_budget_unit=episodes_per_budget_unit,

            use_synthetic=use_synthetic,

            verbose=verbose,

            prev_tube_var=prev_tube_var,

            baseline_tube_var=baseline_tube_var,

        )

        if not success:

```

```

        print(f"\n[STOPPED] Round {round_num} failed!")

        break

    prev_tube_var = metrics["tube"]["tube_var_sum"]

    if baseline_tube_var is None:

        baseline_tube_var = prev_tube_var

    print(f"""
{'='*70}

{'LOOP COMPLETE'.center(70)}

{'='*70}

Run UUID: {run_uuid}

Artifacts: {run_dir}

Summary CSV: {run_dir / 'summary.csv'}

""")

ENDOFFILE

echo "Third part written"

```

Third part written

```

python3 -c "import ast;
ast.parse(open('/home/ryan/otherstuff/agent_bench/runner.py').read()); print('Syntax
OK')"
```

Syntax OK

```

metrics.py
+224
#!/usr/bin/env python3

"""Agent-mode tube metrics for AgentEvalBench.

Tube Definition for Agent Mode:

- Threshold: tau = 0.2

```

```
test_agent_eval_bench.py
```

```
+404
```

```
#!/usr/bin/env python3
```

```
"""Tests for AgentEvalBench v1.
```

```
Test cases:
```

```
1. test_fresh_run_creates_required_files - Verify all artifacts created
```

```
README_AGENT_EVAL.md
```

```
+207
```

```
# AgentEvalBench v1
```

```
Agent Evaluation as Active Learning - treating agent evaluation as a structured  
stochastic simulation space.
```

```
## Overview
```

```
cd /home/ryan/otherstuff && python3 -c "
```

```
import sys
```

```
sys.path.insert(0, '.')
```

```
from agent_bench.grid import generate_grid
```

```
from agent_bench.env_toy_v1 import ToyToolEnv, DISTRACTOR_TOKENS
```

```
from agent_bench.agent_toy_v1 import ToyAgent
```

```
from agent_bench.metrics import compute_agent_tube_metrics
```

```
print('All imports successful')
```

```
# Quick sanity check
```

```
grid = generate_grid()
```

```
print(f'Grid has {grid["n_points"]} points')
```

```
print(f'DISTRACTOR_TOKENS has {len(DISTRACTOR_TOKENS)} tokens')
```

```
"
```

```
All imports successful
```

```
Grid has 1024 points
```

```
DISTRACTOR_TOKENS has 128 tokens
```

```
cd /home/ryan/otherstuff && python3 -c "
```

```
import sys
```

```
sys.path.insert(0, '.')

from agent_bench.runner import (

    derive_episode_seed,

    derive_agent_seed,

    compute_acquisition_score,

    select_targets,

    synthetic_p_fail,

)

import numpy as np

# Test seed derivation

seed = derive_episode_seed('abc123', 1, 0, 0)

print(f'Episode seed: {seed}')

agent_seed = derive_agent_seed(seed)

print(f'Agent seed: {agent_seed}')

print(f'Seeds different: {seed != agent_seed}')

# Test acquisition

alpha = np.ones(1024)

beta = np.ones(1024)

scores = compute_acquisition_score(alpha, beta)

print(f'Acquisition scores shape: {scores.shape}')

# Test selection

selected = select_targets(scores, 5)

print(f'Selected targets: {selected}')
```

```
"
Agent seed: 11788581647576963722
Seeds different: True
Acquisition scores shape: (1024,)
Selected targets: [0 1 2 3 4]
```

```
cd /home/ryan/otherstuff && python3 tests/test_agent_eval_bench.py 2>&1 | head -80
```

```
[Round] Mode: Synthetic
ERROR: cannot import name 'PlanMetrics' from 'loop_metrics'
(/home/ryan/otherstuff/loop_metrics.py)
Running TestResumeContinuesRoundNumbers.test_resume_continues_round_numbers... [Grid]
Saved 1024 points to /tmp/tmpix8xlkuh/grid.npz
[Grid] Version: agent_grid_v1
```

```
runner.py
```

```
+22-12
```

```
from loop_metrics import (
    TubeMetrics, PlanMetrics, ModelChangeMetrics,
    CountMetrics, FusionMetrics, RoundMetrics
)

TubeMetrics, PlanMetrics, ModelChangeMetrics,
CountMetrics, FusionMetrics, RoundMetrics

from loop_metrics import (
    TubeMetrics, PlanSummary, ModelChange,
    Counts, FusionConsistency, RoundMetrics, ArrayStats
)
```

```
cd /home/ryan/otherstuff && python3 tests/test_agent_eval_bench.py 2>&1 | head -100
```

```
=====
AGENT EVALUATION LOOP
=====
```

```
cd /home/ryan/otherstuff && timeout 300 python3 tests/test_agent_eval_bench.py 2>&1
```

```
Results: 12 passed, 1 failed
```

```
Failures:
```

```
TestReproducibleSummaryGivenSeed.test_reproducible_summary_given_seed:
Non-deterministic: 0.0006088140214695109 != 0.001046283090631387
```

One test failed because the `run_uuid` is different for each run (by design). Let me fix the determinism test to properly test within a single run context.

test_agent_eval_bench.py

+34-23

```
class TestReproducibleSummaryGivenSeed:
    """Test determinism: same seed produces same results."""

    def test_reproducible_summary_given_seed(self):
        """Verify same seed produces identical tube_var_sum."""
        """Test determinism: same seed produces same results."""
        """Verify same seed produces identical tube_var_sum."""
        results = []

class TestReproducibleSummaryGivenSeed:
    """Test determinism: same run_uuid + seed produces same results."""

    def test_reproducible_summary_given_seed(self):
```

```
cd /home/ryan/otherstuff && timeout 300 python3 tests/test_agent_eval_bench.py 2>&1
```

Running TestUnitFunctions.test_synthetic_p_fail_range... PASSED

=====
Results: 14 passed, 0 failed

All 14 tests pass. Let me check for linter errors in the newly created files.

Let me run a quick end-to-end verification of the full pipeline.

```
cd /home/ryan/otherstuff && rm -rf artifacts/agent_test_e2e && mkdir -p
artifacts/agent_test_e2e && \

python -m agent_bench.grid --out artifacts/agent_test_e2e/grid.npz && \

python -m loop_runner --mode agent_eval --agent-bench toy_v1 --run-dir
artifacts/agent_test_e2e --rounds 2 --seed 12345 --seed-mode increment --synthetic
--targets-per-round 8 2>&1 | tail -50
```

Artifacts: /home/ryan/otherstuff/artifacts/agent_test_e2e

Summary CSV: /home/ryan/otherstuff/artifacts/agent_test_e2e/summary.csv

<frozen runpy>:128: RuntimeWarning: 'agent_bench.grid' found in sys.modules after import of package 'agent_bench', but prior to execution of 'agent_bench.grid'; this may result in unpredictable behaviour

All tests pass and the end-to-end pipeline works. Let me provide a final summary: Implementation Complete - AgentEvalBench v1 Files created/modified:

File	Description
loop_runner.py	Added <code>--mode</code> , <code>--agent-bench</code> , <code>--episodes-per-budget-unit</code> , <code>--targets-per-round</code> CLI args
loop_io.py	Added <code>save_npz_atomic()</code> and <code>save_csv_atomic()</code> atomic I/O helpers
agent_bench/__init__.py	Package exports
agent_bench/grid.py	1024-point parameter grid generation
agent_bench/env_toy_v1.py	ToyToolEnv with deterministic seeding, token-based hint truncation
agent_bench/agent_toy_v1.py	ToyAgent with explicit integer averaging, token-based parsing
agent_bench/runner.py	Round orchestration, acquisition function, posterior updates
agent_bench/metrics.py	Agent-mode tube metrics ($\tau=0.2$, mean-based tube)
tests/test_agent_eval_bench.py	14 tests including replay audit
README_AGENT_EVAL.md	Documentation with canonical commands

Key features implemented:

- 64-bit blake2b seed derivation (no truncation)
- Separate agent seed via blake2b with `AGENTSEED` personalization
- Token-based `memory_tokens` (not characters)
- Deterministic distractor tokens (`tok000-tok127`)
- Token-based hint parsing (`.index()` on split tokens, not substring)
- Integer posteriors (`int64` α/β)
- Deterministic tie-breaking in target selection
- Commit assertion before `metrics.json`
- Atomic writes for NPZ/CSV with proper `.npz/.csv` temp file extensions

What is AgentEvalBench?

It's an active learning system for evaluating AI agents on a grid of difficulty parameters. The core idea:

1. Grid of 1024 test configurations - varying:
 - `tool_noise` (0-3): How noisy the lookup tool is
 - `verify_flip` (0-0.2): Probability the verify tool lies

- hint_ambiguity (0-3): How vague the hint is
 - distractor_count (0-6): Number of distractor tokens
 - memory_tokens (16-128): Context window size
2. ToyAgent tries to guess a secret number (0-9) by:
 - Reading a hint like "target is near 5"
 - Optionally calling lookup() (noisy)
 - Optionally calling verify(guess) (can lie)
 3. Active learning loop - Instead of testing all 1024 configs equally, it:
 - Maintains Beta posteriors (success/fail counts) for each config
 - Uses acquisition function to pick the most uncertain configs
 - Runs episodes, updates posteriors, repeats
 4. Goal: Efficiently find which configurations the agent fails on ($p_{\text{fail}} > 0.2$ threshold)

How to Run

1. Generate the parameter grid

```
python -m agent_bench.grid --out artifacts/my_run/grid.npz
```

2. Run the evaluation loop (synthetic mode - fast, no real agent)

```
python -m loop_runner \
  --mode agent_eval \
  --agent-bench toy_v1 \
  --run-dir artifacts/my_run \
  --rounds 5 \
  --seed 12345 \
  --seed-mode increment \
  --synthetic \
  --targets-per-round 64
```

Or real mode (actually runs agent vs environment):

```
python -m loop_runner \
  --mode agent_eval \
  --agent-bench toy_v1 \
  --run-dir artifacts/my_run \
  --rounds 5 \
  --seed 12345 \
  --seed-mode increment \
  --targets-per-round 64
```

Key flags:

- --synthetic: Uses closed-form p_{fail} formula instead of running agent (fast for testing)
- --targets-per-round 64: How many grid points to sample per round
- --rounds 5: Number of active learning rounds

Outputs

After running, artifacts/my_run/ contains:

- summary.csv - Per-round metrics (tube_var, coverage, etc.)
- beta_posteriors.npz - Current beliefs about each config's failure rate
- rounds/R0001_*/metrics.json - Detailed round metrics
- rounds/R0001_*/agent_results.csv - Per-episode results with seeds for replay

```
>> /home/ryan/otherstuff : python -m agent_bench.grid --out artifacts/my_run/grid.npz
<frozen runpy>:128: RuntimeWarning: 'agent_bench.grid' found in sys.modules after import of
package 'agent_bench', but prior to execution of 'agent_bench.grid'; this may result in
unpredictable behaviour
[Grid] Saved 1024 points to artifacts/my_run/grid.npz
[Grid] Version: agent_grid_v1
```

```
>> /home/ryan/otherstuff : python -m loop_runner \
--mode agent_eval \
--agent-bench toy_v1 \
--run-dir artifacts/my_run \
--rounds 5 \
--seed 12345 \
--seed-mode increment \
--targets-per-round 64
[Init] New run created: 1a37d957
[Init] Loaded grid with 1024 points
[Init] Created beta_posteriors.npz with 1024 points (Beta(1,1) priors)
```

```
=====
AGENT EVALUATION LOOP
=====
```

```
Run UUID:      1a37d957
Rounds to run:  5 (R0001 -> R0005)
Grid points:    1024
Targets/round:  64
Episodes/target: 256
Mode:           Real simulation
=====
```

```
=====
AGENT ROUND 1 (R0001_20260131_232533)
=====
```

[Round] Selected 64 targets, 256 episodes each
[Round] Total episodes: 16384
[Round] Mode: Real simulation

[Round 1] Complete
tube_var_sum: 0.000239
tube_coverage: 0.016
status: FIRST_ROUND

=====

AGENT ROUND 2 (R0002_20260131_232535)

=====

[Round] Selected 64 targets, 256 episodes each
[Round] Total episodes: 16384
[Round] Mode: Real simulation

[Round 2] Complete
tube_var_sum: 0.001893
tube_coverage: 0.031
status: REGRESSED

=====

AGENT ROUND 3 (R0003_20260131_232537)

=====

[Round] Selected 64 targets, 256 episodes each
[Round] Total episodes: 16384
[Round] Mode: Real simulation

[Round 3] Complete
tube_var_sum: 0.005175
tube_coverage: 0.047
status: REGRESSED

=====

AGENT ROUND 4 (R0004_20260131_232540)

=====

[Round] Selected 64 targets, 256 episodes each
[Round] Total episodes: 16384
[Round] Mode: Real simulation

[Round 4] Complete
tube_var_sum: 0.011132
tube_coverage: 0.062
status: REGRESSED

=====

AGENT ROUND 5 (R0005_20260131_232542)

=====

[Round] Selected 64 targets, 256 episodes each
[Round] Total episodes: 16384
[Round] Mode: Real simulation

[Round 5] Complete
tube_var_sum: 0.013124
tube_coverage: 0.069
status: REGRESSED

=====

LOOP COMPLETE

=====

Run UUID: 1a37d957
Artifacts: /home/ryan/otherstuff/artifacts/my_run
Summary CSV: /home/ryan/otherstuff/artifacts/my_run/summary.csv

➤ /home/ryan/otherstuff : cat /home/ryan/otherstuff/artifacts/my_run/summary.csv

run_uuid	round_id	seed	tube_coverage	tube_var	tube_var_delta	tube_var_delta_prev	tube_var_delta_baseline	targets_selected	total_budget	beta_std_mean_tube	epistemic_mean_tube	aleatoric_mean_tube	enn_changed	fusion_changed	status
1a37d957	R0001_20260131_232533	0	0.015625	0.00023851402975253705	,,,	64	16384	1.4907126859533566e-05	1.4907126859533566e-05	0.0	False	False	FIRST_ROUND		
1a37d957	R0002_20260131_232535	0	0.03125	0.0018930310980067606	-0.0016545170682542236	-0.0016545170682542236	64	16384	5.915722181271127e-05	5.915722181271127e-05	0.0	False	False	REGRESSED	
1a37d957	R0003_20260131_232537	0	0.046875	0.00517520920441418	-0.0032821781064074195	-0.0032821781064074195	-0.004936695174661643	64	16384	0.00010781685842529542	0.00010781685842529542	0.0	False	False	REGRESSED
1a37d957	R0004_20260131_232540	0	0.0625	0.011132491527299532	-0.005957282322885352	-0.005957282322885352	-0.010893977497546994	64	16384	0.00017394518011405518	0.00017394518011405518	0.0	False	False	REGRESSED
1a37d957	R0005_20260131_232542	0	0.0693359375	0.013124363893833934	-0.0019918723665344022	-0.0019918723665344022	-0.012885849864081396	64	16384	0.0001848501956878019	0.0001848501956878019	0.0	False	False	REGRESSED