

Starter Kit for Testing Agentic Systems for Safety and Reliability

Version 1.0
October 2026

Contents

Executive summary	3
1 Introduction	5
1.1 What is this Starter Kit?	5
1.2 Why is agentic testing different?	6
2 What types of agents are there?	8
2.1 Basics of agents and agentic systems	8
2.2 Common enterprise deployments today	9
3 The four steps of agentic testing.....	12
3.1 Identify the relevant risks for testing	12
3.2 Create realistic and robust tests	17
3.2.1 Setting up the testing environment and harness	18
3.2.2 Setting up logging and observability	21
3.2.3 Defining test tasks, metrics and evaluators.....	23
3.3 Run tests and analyse logs	29
3.3.1 Running tests	29
3.3.2 Analysing results and logs	30
3.4 Continuously monitor and re-test	32
4 Where to go from here	33
4.1 Challenges and open questions	33
4.2 Future work	33
5 Case studies.....	34
5.1 Case Study: RFP requirements extraction agents (Cohere)	34
5.2 Case Study: Code-fix agent (CyberSierra)	38
5.3 Case Study: SDLC-support agents (KBTG)	41
5.4 Case Study: Financial reconciliation agent (EarlyBird AI & Knovel Engineering).....	43
5.5 Case study: Content verification agent (IMDA)	45
Annex A: Other useful resources	48
Annex B: Call for feedback	49
Acknowledgements.....	50

Executive summary

Organisations are increasingly deploying agentic AI systems across business functions, from customer support to internal operations. Unlike traditional LLM apps that primarily generate content, agents act on their environment with limited human involvement. A customer service agent can issue refunds, not just answer questions; a coding agent can search codebases for vulnerabilities and decide which files to edit. Malfunctions can therefore translate into real impact.

Testing is essential to ensure that these agentic systems perform safely and reliably in the real world, especially as they are non-deterministic in nature. It is part of the third dimension on implementing technical controls and processes in the [Model AI Governance Framework for Agentic AI](#), ensuring that controls are effective and agentic systems behave as expected.

This Starter Kit provides a practical reference to help enterprises get started with agentic testing. It consolidates emerging good practices and lays out a structured approach to testing agentic AI systems for known risks. The guidance draws on real-world testing through the [Global AI Assurance Sandbox](#), engagements with industry deployers and third-party testers, and practitioners from the Cyber Security Agency of Singapore (CSA) and the Government Technology Agency of Singapore (GovTech).

The focus of this Starter Kit is on testing for safety and reliability. This includes risks such as erroneous or unauthorised actions, data breaches, and biased or unfair actions, which may be caused by routine use or malicious actors. While broader performance and cybersecurity are not the primary focus of this version of the Starter Kit, references are included where relevant.

The guidance introduces key concepts and explains how agentic testing differs from other AI systems. While software and LLM testing practices continue to apply, agentic systems have additional testing needs because they can take actions over multiple steps with limited human involvement. These include testing new components such as tool use, and testing over the agent's entire trajectory rather than just its output. Such testing is enabled by deeper observability into the agent's process, including its interactions with tools, other agents and human approvals.

The Starter Kit lays out a four-step process recommended for testing agentic systems:

- 1. Identify the relevant risks for testing.** Risk assessment and mitigation begin during system design (*safety by design*), but residual risks remain, and safeguards may not always work as intended. Common risks affecting agentic systems include erroneous actions, unauthorised actions, biased or unfair actions, data breaches, and disruption to connected systems. While human oversight is a common safeguard, inadequate or uninformed oversight may itself become a risk. For testing, translate risks into specific scenarios based on your use case and implementation, e.g. what constitutes a data breach per your organisation's policy or local law. Prioritise scenarios for testing based on their likelihood and potential impact.
- 2. Create realistic and robust tests.**
 - a. Set up test environment and harness:** As an agentic system can dynamically respond to the external environment, there is a greater need to test it in real-world

conditions to understand its spectrum of behaviour, but this should be balanced against its corresponding potential to cause real-world harm. Start in contained but realistic environments, moving closer to production environments when there is greater confidence.

- b. **Set up logging mechanisms:** Capture the agent's trajectory, including relevant actions, tool calls, interactions, state changes and outcomes, so that behaviour can be reconstructed and assessed. Logs should be designed around the risks and behaviours being tested, and structured where possible to support efficient parsing and analysis.
- c. **Design tasks, metrics, and evaluators:** The principles of creating realistic test tasks still apply. Create tasks that test for specific risks, but also at different levels (e.g. for individual components like memory). However, in general, (a) invest more effort in creating realistic tasks, as agents are sensitive to indications that they are in simulations (e.g. names like "John Doe"), and may act differently in such cases; and (b) unlike LLM apps, users tend to engage agentic systems over interactive, longer-horizon tasks. Test data should reflect that. In terms of metrics, some new metrics specific to agentic systems include tool use accuracy and handoff accuracy.

3. Run tests and analyse logs.

- a. **When deciding on the number and extent of test runs,** some useful parameters to consider include the number of repeated runs (to account for variability in agent behaviour), different model-harness combinations (where models can be swapped dynamically by the user), and the size of the context window (compaction may cause agentic systems to lose previous instructions).
- b. **When analysing the results,** log analysis helps understand why an agent behaved the way it did. An agent may produce the right outcome while taking incorrect actions or violating boundaries along the way. However, agentic logs can be unstructured and voluminous, including each agent's reasoning, tool calls and interactions with users. Analysing every aspect of every trajectory at scale remains challenging. It helps to start small and progressively automate the analysis, such as identifying what the most important signals of failure modes are, using these to develop automated checks or scanners, and incorporating some level of human analysis.

- 4. **Continuously monitor and re-test.** All AI systems can drift over time, but agentic systems introduce additional sources of change through tool and service integrations, memory, and long-horizon interactions. Monitor the agentic system continuously and re-test periodically post-deployment, including the effectiveness of human oversight over the system.

Agentic testing remains a fast-moving area, with methodologies still evolving and practical challenges remaining. This Starter Kit provides a starting point based on current and emerging best practices and will continue to evolve as testing approaches mature.

1 Introduction

1.1 What is this Starter Kit?

This guidance is a practical reference for enterprises that collates emerging good practices for testing agentic systems. For those familiar with testing software or LLM apps, it highlights what is new about agentic testing, such as testing tool use or analysing agent trajectories. For those relatively new to testing, it maps out a structured approach covering risks, test design, log analysis and monitoring. After going through this guidance, readers should have a better understanding of how to get started with agentic testing.

It also includes [case studies](#) on testing actual agentic deployments. Agentic systems can differ widely in their use case, action-space and autonomy, changing their risk surface and what needs to be tested. For instance, testing a customer service agent with access to enterprise data may focus on whether it follows company policies and handles sensitive data appropriately, while testing a coding agent may focus on code quality and safety. Case studies are intended to provide some illustration of what contextualising testing to a specific agentic implementation can look like.

The focus of this guidance is on testing for *safety and reliability*. We focus on common risks affecting agentic systems, namely erroneous or unauthorised actions, data breaches, biased or unfair actions, and disruption to connected systems. This can overlap with functionality testing insofar as correctness affects a system's safety and reliability. However, it does not focus on other performance-related concerns such as cost and latency.

Security is also a core concern for agentic systems.¹ Agentic systems are exposed to a greater attack surface as they interact with external systems and may be compromised through attacks such as memory poisoning or indirect prompt injections through tool use. While this guidance covers various security considerations, it does not provide an exhaustive treatment of cybersecurity risks or testing. Threat modelling and risk assessment therefore remain important for identifying relevant attack vectors and informing security testing and appropriate controls. Organisations may refer to CSA's Guidelines and Companion Guide on Securing AI Systems, together with the [Addendum on Securing Agentic AI](#), for more detailed guidance on identifying, assessing and mitigating such risks based on the capabilities of the system.

This guidance is geared towards enterprise deployments. It is intended for enterprise developers, testing teams and third-party testers conducting agentic testing, and compliance and responsible AI professionals seeking to understand and assess their organisations' testing practices.

It also builds on existing guidance on AI governance and testing.

- The [Model AI Governance Framework for Agentic AI](#) (MGF for Agentic AI) sets out a broader approach to responsible agent deployment. It recommends technical controls across the agent lifecycle, including testing. This Starter kit provides practical guidance on such testing.

¹ See [OWASP Top 10 for Agentic Applications 2026](#), [CSA's Addendum on Securing Agentic AI](#) and [GDM AI Control Roadmap](#).

- The [Starter Kit for Testing of LLM-based Applications for Safety and Reliability](#) (LLM App Testing Starter Kit) provides detailed guidance on LLM testing. This document builds on those foundations to address the additional testing needs of agentic systems.

1.2 Why is agentic testing different?

As agentic systems are fundamentally software built on LLMs, software and LLM testing principles still apply. This includes unit and integration testing for software systems, and selecting representative datasets, metrics and evaluators.

However, agentic systems introduce additional testing needs. Such systems can act over multiple steps with varying degrees of human involvement, use tools and interact with external systems. These capabilities can extend existing risks and introduce new failure modes like erroneous or unauthorised actions, misuse of tools or permissions, failure to seek human approval and security compromises. The specific risks depend on the system’s capabilities, autonomy and use case, and are discussed further in [Section: Identify the relevant risks for testing](#).

This informs both how the test is set up and what needs to be tested.

1. How the test is set up: Creating realistic and robust test environments

Account for additional components like tools and memory across interactions. Agents use tools to interact with external systems. They also use memory, which is not a new component, but is increasingly relevant due to their ability to complete longer-horizon tasks over multiple turns. These components introduce additional failure modes and attack surfaces. For instance, tools may fail due to hitting rate limits, agents may call them with the wrong inputs, and attackers may compromise them by poisoning tool descriptions. Test setup and design should account for these risk scenarios. [Section: Testing specific components – memory and tools](#) covers this in detail.

Use realistic environments while containing real-world harm. Testing in environments close to production can make results more meaningful and uncover behaviours that simplified settings may mask. They also help prevent evaluation or simulation awareness,² which may compromise the test. At the same time, because agents can act on their environment, there is a need to minimise real-world impact from failures. [Section: Setting up the testing environment and harness](#) recommends a progressive approach from simulated or sandboxed environments towards real systems, with appropriate controls and oversight.

2. What needs to be tested: What the agent thought, did, and how it got there

Test actions. For LLM apps, the focus was largely on the quality and safety of the generated content. In contrast, agentic systems take actions, which may not be visible in the eventual output, but also need to be verified. For example, if the task is to send an email and schedule

² This refers to the phenomenon where models are aware that they are in a simulated environment and/or being evaluated, which can affect their behaviour. See for example [UK AISI, GPT-6 performs unsanctioned supply-chain attacks in simulations](#).

a call, the test must confirm not only that the agentic system stated that it executed the tasks, but also that the email was sent and the call was scheduled.

Test trajectories. Agentic systems generally act over multiple steps with some level of autonomy. Agents may complete the same task through different execution paths, and a correct outcome does not necessarily mean the process was acceptable. Tests should therefore assess how the task is completed, including whether the agent uses appropriate tools, processes tool outputs correctly, hands off to other agents at the right points with sufficient context, and follows required processes. See [Section: Setting up logging and observability](#).

Test boundaries. Agents are trained to complete tasks, and may apply unanticipated solutions or try workarounds to get around technical controls to achieve their goals.³ It is important to test that controls and boundaries⁴ hold, especially when agents with different permission levels are chained together in the same workflow.

Test approvals, coordination and handoffs. Many agentic systems rely on human approvals before certain actions can proceed. They may also need to coordinate and hand off to other agents. Capturing and analysing these interactions is important.

These differences also make test tasks, metrics and analysis more complex. Tasks may involve multi-turn interactions and multiple failure modes to test for. Success criteria need to account for multiple valid execution paths and metrics need to capture both outcomes and behaviour.

While pre-deployment testing is important and should be carried out, there is a limit to what it can do, especially for agentic systems. Agentic systems tend to be designed to receive and react to real-time feedback from external systems. As tools or their underlying APIs get refreshed or fail, or underlying models drift,⁵ agents may have to handle situations that pre-deployment testing did not cater for. Post-deployment monitoring of behavioural drift, incidents and near-misses is necessary to identify unexpected behaviours, respond to emerging risks and maintain effective safeguards. Re-testing at appropriate junctures (e.g. when there are material updates to models, guardrails or integrations) is also important.

³ See for example the recent [OpenAI-HuggingFace hacking incident](#), where one of the “key driver[s] of the misaligned behaviour was that the agents rarely ‘gave up’ on their evaluation tasks, even when the tasks appeared impossible to solve”.

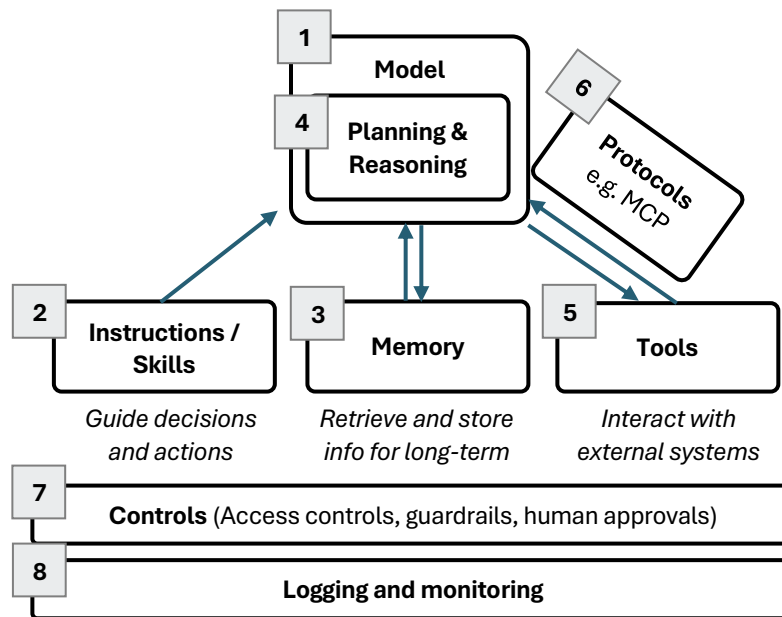
⁴ See Business Insider, [Meta’s Muse agent sharing a user’s address with a buyer](#) and Australian Government, [OpenAI agent incident involving unauthorised access to the Medicare statistics portal](#).

⁵ Model drift refers to the degradation of AI performance over time, as illustrated in IBM, [What is Model Drift](#).

2 What types of agents are there?

This Starter Kit focuses on enterprise deployments and testing needs in industry today. Agentic systems sit on a spectrum of autonomy and action-space. Today, many enterprises deploy more bounded, low-autonomy agentic systems with tightly scoped objectives and access, even when agents can orchestrate or plan. However, more unbounded implementations are emerging.

2.1 Basics of agents and agentic systems



Core components of a simple agent

An AI agent is built from several core components.⁶ Deployers of LLM apps may be familiar with some components such as the model (usually an LLM), instructions, and memory.⁷ The model used for an agent enables it to plan and reason. Tools allow agents to interact with external systems, protocols structure such interactions, and instructions and/or skills guide which actions to take and which tools to use. Finally, there are components to enable safety and reliability, such as controls to constrain the agent's behaviour, and logging and monitoring for traceability and interventions.

Agentic systems refer to the entire system being deployed and can include one or more agents. Multi-agent systems are useful where tasks decompose functionally and allow you to define clearer rules for each agent based on its specific functionality. In workflow automation, multiple agents can

⁶ For a more comprehensive overview on what agents and agentic systems are, see [MGF for Agentic AI](#). See also Langchain, [The Anatomy of an Agent Harness](#).

⁷ Agents use more long-term memory for long-horizon workflows and greater personalisation. Such memory can be episodic (events, actions and outcomes), semantic (mappings, facts, ontologies) and/or procedural (learned skills). These may be stored in databases or collated as skills and markdown files.

be set up such that one agent completes a portion of the task before handing off sequentially to the next agent. For more flexible workflows, agents may call each other, with different kinds of coordination, handoffs and quality-checking mechanisms.

What components an agentic system includes and how the agentic system is architected impacts how it should be tested. A common first step in testing is to lay out the core components and architecture of the system (including its controls) to evaluate where the risks are and what should be tested. For example, where web search tools are used, a common risk is the agent receiving untrusted content from the web. Such content may contain prompt injections that mislead the agent into taking unsafe actions. Whether the agent can identify untrusted content and reject and/or sanitise it should be tested.

The principles for testing single- and multi-agent systems are similar, save for a few notes:

- *Unpredictability and complexity may increase.* Agent behaviour can be non-deterministic, and interactions between multiple agents may increase the number and complexity of possible execution paths. This can lead to emergent behaviours⁸ and cascading errors.⁹
 - However, this ultimately depends on the specific deployment and architecture of the system. For example, chaining agents sequentially, with each agent focused on a single task, may make it easier to implement and enforce scoped access tools on each agent, as well as evaluate each agent independently.
- *Evaluate agents independently and together.* A system of individually safe and reliable agents is not necessarily a safe and reliable system overall.¹⁰ Test each agent independently, to understand their individual failure modes, as well as the end-to-end behaviour of the system for system-level failures. Repeatedly test areas of low determinism for agent consistency and evaluate end-to-end in multi-agent systems.
- *Test coordination mechanisms.* Assess the reliability of coordination mechanisms such as handoffs and communication quality. It is unsafe to assume that agents interact in similar ways as humans, and interaction points between agents should be tested.

2.2 Common enterprise deployments today

Enterprise agentic deployments differ in terms of the action-space and autonomy of each agentic system. Action-space refers to the range of actions the agent can take, commonly determined by the tools it has and permissions on those tools. Autonomy refers to the degree to which an agent can decide how to act towards a goal, such as by defining the steps to take.¹¹

This Starter Kit focuses on two archetypes that reflect common enterprise deployments today: workflow automation and goal-oriented (but still bounded) agentic systems. These are not rigid categories, and a system may exhibit characteristics of both.

⁸ See GDM, [Investing in multi-agent AI safety research](#).

⁹ See Gradient Institute, [Risk Analysis Techniques for Governed LLM-based Multi-Agent Systems](#), and Wharton AI & Analytics Institute, [Governing AI Agents: What the Amazon Outage Reveals about Enterprise Risk](#).

¹⁰ See Australia AI Safety Institute, [Risks and controls for multi-agent systems](#).

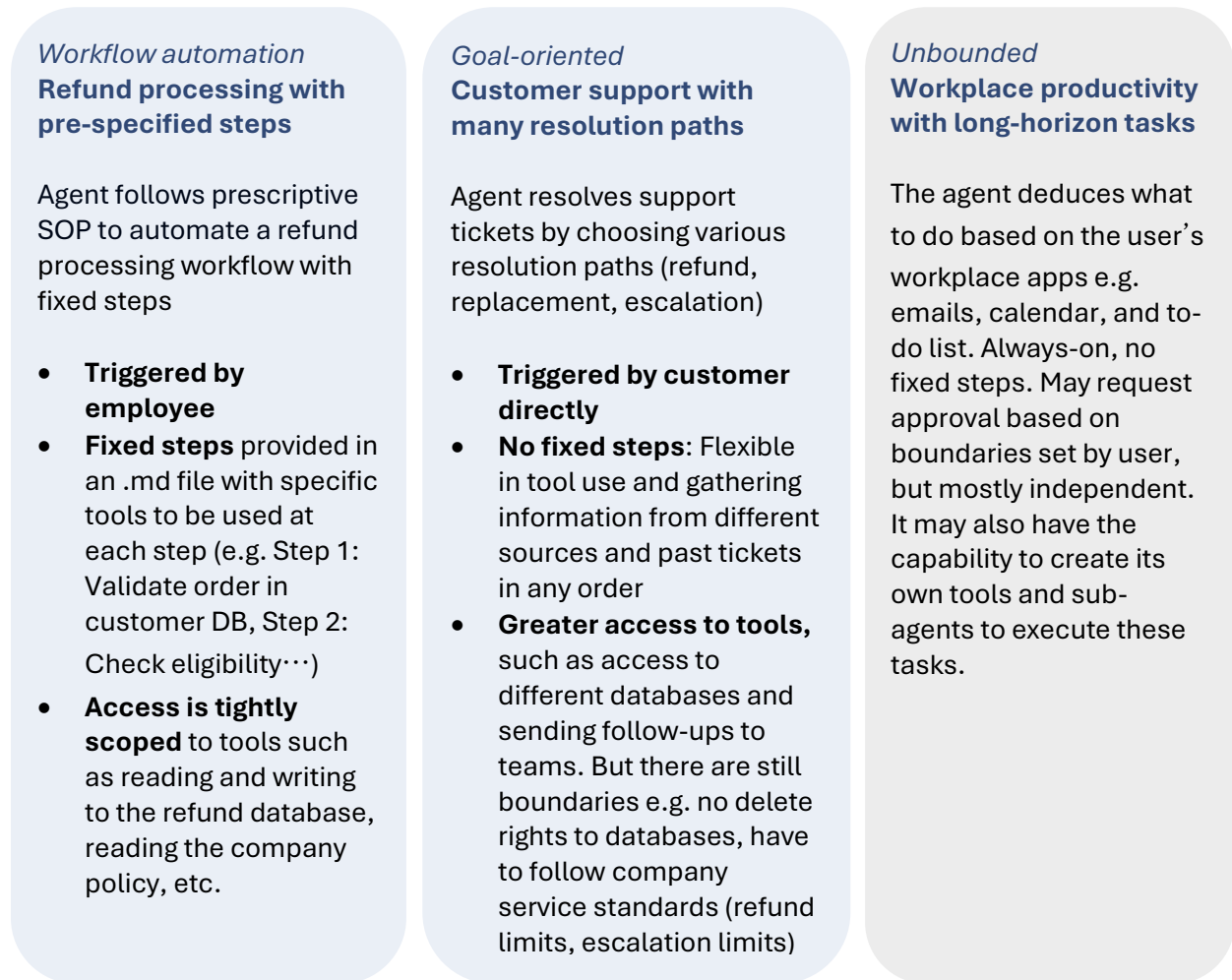
¹¹ See WEF, [AI Agents in Action: Foundations for Evaluation and Governance](#).

Identifying which characteristics your system exhibits provides a useful starting point for understanding its risk scenarios and testing focus.

	More common in enterprise deployments today, can be single or multi-agent		Less common, not the focus of this guidance
	Workflow automation	Goal-oriented	Unbounded, fully autonomous
Common features	• Predefined workflow or SOPs,¹² where agents are expected to follow specific steps. There is limited decision-making around orchestration though there might be decisions within the step. • More prescriptive: Orchestrated by code, system prompt, user instructions and/or policy docs. • Tightly scoped tools and access: Each step may be mapped to the specific tools/skills to use. Access is tight.	• Predefined objectives but less defined workflows – more flexibility to plan and act. • Less prescriptive: Orchestration is guided (and bounded) by instructions, guardrails. • Wider tool choice and access: Ability to choose between a few tools for the same task. Access and privileges are broader but with limits.	• Broader objectives, has ability to set more intermediate goals and may be more proactive. • Minimally prescriptive: Freedom to plan and orchestrate.
Testing focus	Test that agentic system <i>follows the predefined workflow</i> and does not do anything outside of that workflow.	Test that agentic system sticks to <i>objectives</i> and stays within its <i>boundaries</i> (e.g. seek human approval where appropriate).	While the testing principles are similar, testing usually has to be carried out at a much larger scale to ensure that the spectrum of behaviour is captured, and to investigate unintended side-effects or behaviour.
Case studies	5.1 , 5.4 , 5.5	5.2 , 5.3	

¹² See AWS [Agents SOP](#), [Anthropic Skills](#).

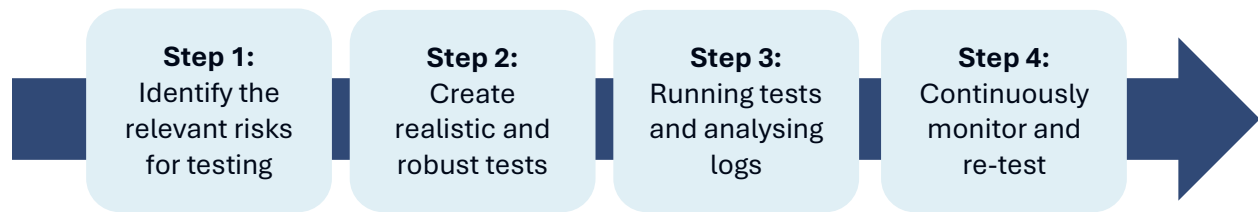
We use the following illustrative use cases to demonstrate each archetype. These examples are also used later in the guidance to illustrate how to contextualise risk assessment.



Example use cases to illustrate different kinds of deployment

The intention is not to place every use case neatly into one bucket, but rather to identify the predominant archetypes of deployment. Many use cases may exhibit a mix of these characteristics, e.g. a series of prescriptively defined steps, while having more flexibility around a certain step, decision or action.

3 The four steps of agentic testing

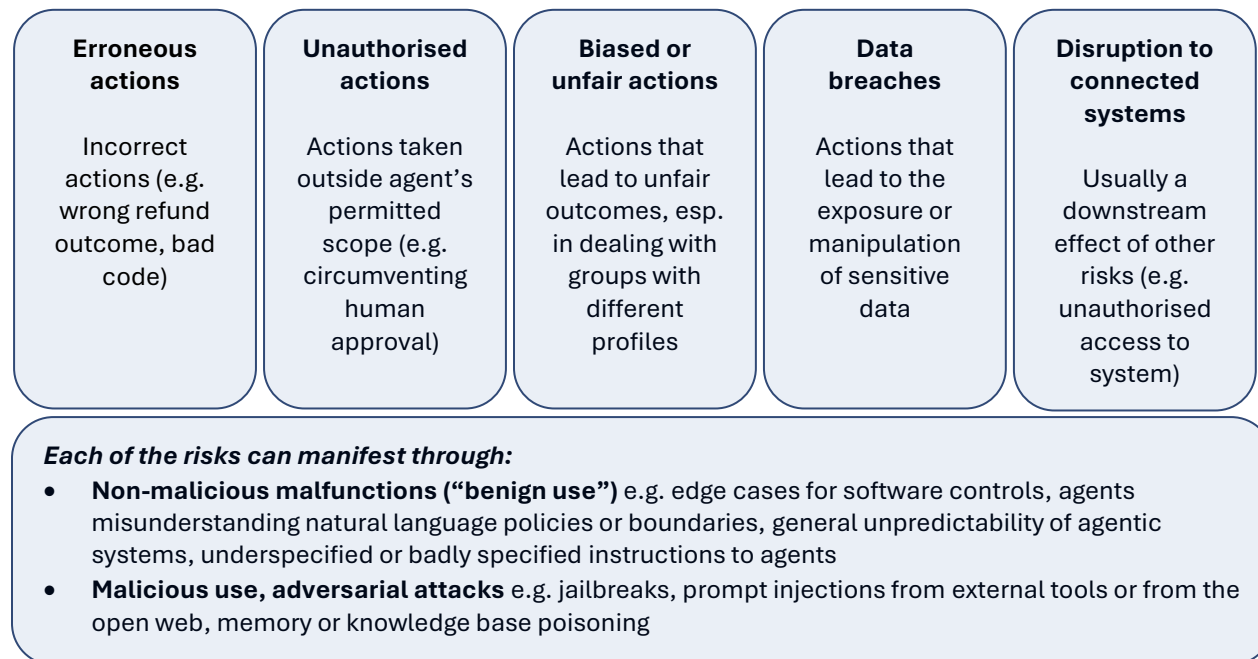


The four steps of agentic testing

We recommend a four-step approach to agentic testing. The approach may be familiar to those used to testing software and LLM apps, with some specific notes for agentic systems.

3.1 Identify the relevant risks for testing

Each agentic system will have its specific risks depending on implementation and use case, but common risks affecting agentic systems are:



Risks of agentic systems¹³

Using the frame above, derive specific risk scenarios and assess their likelihood and impact so that you can prioritise what to test. There are two angles to consider:

- **Agent’s intended purpose, capabilities, failure modes and boundaries**

¹³ These are risks identified in [MGF for Agentic AI](#), Section 1.2.2.

Starting from the agent's intended purpose and capabilities¹⁴ can usually point towards what can go wrong if they are wrongly used or misused. For example, if an agent can access a database, it may be able to update it with the wrong information, delete it, or leak data. Consider:

- What can the agent do?
- What decisions can it make and with what level of autonomy?
- Does it interact with third-party tools, external systems, or users?
- Are the users internal, external, or members of the public?
- Are the agent's actions reversible?

- **How the risks contextualise to the use case, both at outcome and trajectory level**

Agentic implementations tend to be use-case specific. What is a correct trajectory and/or outcome depends on its intended purpose, operational context and legal/regulatory requirements. For example, what constitutes sensitive data and what it means to handle it correctly can vary from system to system. Consider:

- What is a correct outcome?
- What is the correct trajectory or process?
 - Are there steps the agentic system *must* take, such as seeking human approval or consulting a policy document?
- Can wrong processes yield correct outcomes?
 - e.g. if the agent does not consult the policy document but updates the database correctly anyway, would you consider that acceptable?
- What identities, credentials and privileges does the agent possess or inherit? What can it access on behalf of the user? What trust boundaries does it cross?
- What constitutes a data breach or a biased decision? Are there laws or regulatory requirements to be mindful of?
- What guardrails and safeguards have been implemented, how have they been implemented, and how fail-proof are they?
 - e.g. do they rely on instructions (e.g. system prompts), filters, permissions or firewalls/isolation? How a safeguard is implemented can affect how likely it is to be bypassed or fail under different conditions.
- Where can untrusted or attacker-controlled inputs enter e.g. user prompts, retrieved web content, tools?
- What happens if a component is compromised? What is the maximum blast radius?

Factors affecting the likelihood and impact of each risk scenario include the agent's domain and use case, access to sensitive data, access to external systems, reversibility and scope of its actions (e.g. read-only or full access), level of autonomy, task and system complexity, and whether the agent is provided or operated by an external party.¹⁵ On level of autonomy, it is worth noting that an agentic system can have deterministic and non-deterministic elements. For deterministic elements, such as where some parts of the workflow are fully orchestrated by rules-based code, the risk of unpredictability is smaller, although the workflow should still be tested for edge cases.

¹⁴ See GovTech, [Agentic Risk and Capability Framework](#).

¹⁵ See [MGF for Agentic AI](#), Section 2.1.

Security risks for agentic systems

The risks above like unauthorised actions, data compromise, and disruption to connected systems overlap with security risks when triggered by malicious actors.

Hence, the key consideration for security testing is to go beyond representative conditions and foreseeable failures, and to assume that an adaptive adversary is deliberately seeking to identify and exploit weaknesses, bypass controls, chain vulnerabilities and achieve effects outside the system's intended boundaries. As a result, security testing should be threat-informed, and testing should go beyond whether a guardrail works against a predefined attack, checking whether it remains effective when attacks are varied, chained, obfuscated or delivered through different trust boundaries. This makes it more important to understand the risk scenarios involved, including how an attacker may access the system, and test enforceability of the controls.¹⁶

As part of the risk assessment, security risks that should be considered include agent goal manipulation, tool misuse, memory poisoning and privilege compromise¹⁷ among others.

It is important to conduct risk assessment from the get-go, i.e. at the time of selecting the use case and designing the agentic system. This assessment should inform which risks to prioritise for testing. Testing can also help assess the extent of these risks, for example, by estimating how likely they are to materialise under different conditions.

We bring back the example use cases in [Section: Common enterprise deployments today](#) to demonstrate what these risk scenarios may look like. Consider the refund agent example:

Workflow automation: Refund processing with pre-specified steps (<i>Illustrative</i>)	
What it can or cannot do	How it informs risk scenarios
It can process refunds under \$100 without human approval - It needs to follow a 5-step verification process in the sequence specified in its .md file - It must stick to strictly specified tools, with	<u>Erroneous actions</u> - It may make the wrong decision: - False positive: Agent agrees to a refund that violates the refund policy¹⁸ - False negative: Agent refuses a refund permitted by the refund policy - It may skip a verification step. Even if the eventual outcome is compliant, it sets a dangerous precedent - It may check the wrong order or customer information - It may hallucinate customer details - It may claim it has initiated the refund, but did not actually do so

¹⁶ CSA's [Addendum on Securing Agentic AI](#) provides information on threat modelling and taint tracing for agentic systems. Also see [GDM AI Control Roadmap](#).

¹⁷ For a more comprehensive coverage of security threats, see [OWASP, Agentic AI – Threats and Mitigations](#).

¹⁸ Reminiscent of the [Air Canada incident](#), such failures can happen due to outdated or incorrect sources, insufficient steering, or hallucination/misinterpretation of organisational policy. Log analysis can help here.

highly managed permissions It cannot access the web It is employee-facing	<u>Unauthorised actions</u> - It may not seek approval for a request over \$100 - It may try to search the web for customer details (<i>and may succeed if the guardrail doesn't work as intended</i>) <u>Biased or unfair actions</u> - It may preferentially approve some claims or customer profiles over others <u>Data breaches and attacks</u> - Lower risk due to tight access controls and because it is only employee-facing <u>Disruption to connected systems</u> - It may edit the database incorrectly, malforming internal data
---	--

Comparing this with the customer service agent example, many of the risk scenarios above would apply, but there would be additional and distinct ones to consider:

Goal-oriented: Customer support with many resolution paths (<i>Illustrative</i>)	
What it can or cannot do	How it informs risk scenarios
It can decide how to address customer issues and choose appropriate resolution paths - It needs to seek human approval if it cannot find a reasonable resolution path - It can access the web to search for online reviews or reconcile social media complaints, update databases based on its resolutions, send follow-ups to different teams, etc. It is customer-facing	<u>Erroneous actions</u> - It may not be able to identify the correct issue - e.g. fee waiver vs membership termination - It may not make a correct plan <i>This was not a risk for the previous case, as the plan was provided to it and the concern was following it</i> - It may issue a wrong outcome - e.g. rejected request [<i>similar to refund agent</i>] - It may reference historical resolution paths but interpret them wrongly <u>Unauthorised actions</u> - It may not seek human approval even if it cannot find a reasonable resolution path <i>Unlike the previous case, which had a defined approval limit (\$100), it may be harder to define what failing to find a reasonable resolution path looks like</i> - As action space and autonomy expand, more severe possibilities may arise - e.g. it may declare that the customer's request is unfounded, add them to a "do not engage" list or delete them from the company's systems

	<u>Biased or unfair actions</u> • It may act preferentially towards some customer profiles <u>Data breaches</u> • It may accidentally reveal another customer’s details to the current user ○ e.g. justifying a decision with details of a past decision involving another user • Session resets not done properly, leaking data from past sessions with different users <u>Attacks</u> • As it is external-facing and interfaces with the web, it may be poisoned by webpages, affecting tool use, retrieval, or memory • User may try to jailbreak it and ask it to give unreasonable refunds, delete customer databases, or upload poisoned documents <u>Disruption to connected systems</u> • It may access external systems, e.g. customer review site and crash the website while trying to delete a customer review
--	---

These examples are illustrative but show how risks can be translated into specific scenarios contextualised to the use case. Clearly defined and comprehensive risk scenarios make it easier to develop corresponding test tasks and metrics. Some edge cases will likely remain as systems become more autonomous and complex, but efforts should be made to minimise these, especially for high-risk failure modes.

It is also important to test whether human oversight and controls are truly effective, i.e. whether humans have sufficient visibility and understanding of agent actions and can intervene and stop them where necessary without decision fatigue.

How do the risks in this section link with the [LLM App Testing Starter Kit](#)?

The Starter Kit for LLM apps identifies risks primarily associated with generated content. For agentic systems, these risks continue to apply but take on the additional dimensions of being able to plan, act and interact. The risks in this document reflect this extension:

- **Hallucination and inaccuracy → Erroneous actions:** Inaccuracies can translate into incorrect tool calls, fabricated actions or incorrect claims about actions taken.
- **Undesirable content → Erroneous / unauthorised actions:** Agents may still generate harmful or toxic content, which would be in violation of their guiding policy, and may translate into harmful actions and outcomes.
- **Data leakage → Data breaches:** Agents’ access to tools and data can turn information leakage into unauthorised access, transfer or manipulation of data.

- **Bias in decision making → Bias in actions and decisions:** A biased recommendation or assessment can turn into an unfair action.
- **Adversarial prompts → Broader attack surfaces:** Beyond direct prompt injection, agents may be exposed to indirect prompt injection and attacks through tools, memory and other components.

3.2 Create realistic and robust tests

This section introduces the key components of an agentic test setup and covers good practices for designing these components. These components are inherently linked. The use case and purpose inform the test environment, harness, logging and observability mechanisms capture the agent's behaviour, and test tasks and metrics are designed to assess relevant risk scenarios.



Main components of an agentic test setup

The main components for setting up agentic tests are:¹⁹

- **Test environment:** This is the world that the agent is tested in and includes all the external systems that the agent can access, usually through its tools. It includes MCP servers and tools the agent calls, as well as the underlying data and systems accessed by those tools. For instance, a filesystem MCP server can provide tools such as `read_file(file_x)` and `write_file(file_x)`, whereas the underlying data and systems can refer to the file directory in which `file_x` is located and the contents of `file_x`. This can also extend to other agents that

¹⁹ See also Anthropic, [Demystifying evals for AI agents](#).

it may connect with, such as by using the A2A protocol.²⁰ This “world” can be simulated or located within development, staging or production environments.

- **The human:** Where feasible, testing the overall workflow (including the human components, or at the very least, simulated users) is critical to getting a clear understanding of how the system will perform when the agent is deployed within it. It can help uncover finer points such as decision fatigue, approval-review backlogs (human becomes a bottleneck) and effectiveness of human oversight.
- **Test harness:** A test harness orchestrates the testing pipeline, which includes running the tests, resetting it between runs, and recording test logs and results. Its primary function is running the test: launching and feeding context, tasks and available tools to the agents, managing the runs, and scoring performance.

Between test runs, you may need to reset the test environment to its original state. For example, if an agent issued a refund in one run, it may have also updated the corresponding refund status in the customer database. This would have to be reset before the next run. Finally, as the tests are run, the test harness also logs each plan, step, and action taken by the agentic system, including any policy checks or guardrails that are activated. This enables detailed analysis of the agentic trajectories and results after the test.

- **Test tasks, evaluators, and metrics:** Test tasks, evaluators and metrics are broadly similar to testing LLM apps. The *test tasks* would usually be a user request to complete a task e.g. analyse a certain dataset. There is a broader spectrum of *metrics* when testing agentic systems apart from overall accuracy and safety of the outcome, such as tool call accuracy (whether the right tools were called with the right inputs) and task adherence (whether agent stuck to the task). The *evaluators* can range from rule-based scripts (e.g. that check if certain tools were called) to LLM judges (e.g. to check if the agent complied with a natural language policy). More details are provided in [Section: Defining test tasks, metrics and evaluators](#).
- **Agentic system:** While not part of the test setup itself, its design affects how testing should be configured. For example, exposed reasoning or decision traces should be logged, while the model’s context window can affect how the harness supplies information during testing.

3.2.1 Setting up the testing environment and harness

Testing should be as realistic as possible, while ensuring appropriate containment based on the system’s potential impact. Realistic testing helps ensure that the findings meaningfully reflect behaviour in production settings, while appropriate containment controls limit the potential impact if failures or successful adversarial attacks occur during testing.

A progressive approach can help achieve this balance. Testing can begin in simulated or sandboxed environments, and move closer to production as confidence in the system grows.

²⁰

See [A2A Protocol](#).

- **Start contained but realistic.** Testing may begin in simulated²¹ or sandboxed environments. But this should not mean oversimplification. Use realistic data, tools and interactions where possible, as agents may be sensitive to obvious signs of evaluation, such as dummy data (e.g. “John Smith”), locally hosted sites and simplistic tools.²² This helps to ensure that the test findings are reflective of real expected behaviour and to avoid evaluation awareness.
- **Move closer to production as confidence in the system increases.** Progressively introduce staging or test systems, limited connections to real systems, and eventually production testing where appropriate. At each stage, consider the tools and connections needed, their potential blast radius, and what safeguards or sandboxing can be retained. Implement multilayered safeguards to ensure fallback mechanisms in case controls fail.
- **Use human oversight and monitoring.** Where agents can make meaningful, consequential changes to the real-world environment, incorporate human oversight and continuous monitoring (e.g. automated alerts) into the test runs to catch any unexpected behaviours.

Good practices for creating *realistic*, but simulated, testing environments include:

- **Environment:** Mirror the production environment as closely as possible.
 - **Mirror architecture** (including tools, MCP servers, services, memory, guardrails, integrations) with realistic tools, parameters (even when tools are duplicated, maintain realism – e.g. if original tool has a docstring, keep it).
 - **Keep environment data realistic**, reflecting the scale, complexity and imperfections of real systems. Avoid overly simple setups, such as a database with only one customer or sparsely populated records, and consider realistic volumes, timestamps and histories in mocked systems such as email servers. Where anticipated in production, include mixed-language inputs and artefacts, messy or unstructured data, poorly formatted records, or images to test robustness under realistic conditions.
 - **Follow industry (or organisation-specific) norms** and realistic URLs/paths, names, markers – e.g. avoid placeholder or dummy data like john.doe@gmail.com, abc@xyz.com or locally hosted sites like http://localhost:8000.
 - **Simulate realistic environment failures**, such as API timeouts or errors, unavailable services, authentication failures, stale information, and malformed tool outputs. These can be deliberately introduced into test cases to assess whether the agent handles failures gracefully and recovers appropriately.

²¹ Agents often process tool inputs and outputs without seeing the underlying functionality of the tool. Hence, tool responses can often be simulated, e.g. outputting a response without actually sending it.

²² See UK AISI, [GPT-6 performs unsanctioned supply-chain attacks in simulations](#), and Singapore and Korea AI Safety Institutes, [Testing AI agents for Data Leakage Risks in Realistic Tasks](#).

- **Testing harness:**
 - **Ensure resets** between runs, especially in stateful testing.²³ Remove shared state (e.g. cached files, previous outputs, interaction history) that could leak information or affect subsequent runs.
 - **Ensure consistent testing parameters** between runs.
- **Multi-turn user interactions**
 - **It is common to simulate a user using another LLM**, especially when testing at scale – in that case, provide detailed instructions to the user-LLM (user persona), and ensure enough information has been given to reduce chances of hallucination.

Good practices for creating *robust and contained* testing environments include:

- **Minimise connections to live systems**, including production infrastructure and external tools, services and websites, especially when testing boundary violations or high-risk jailbreak scenarios.
- **Mock or simulate MCP servers**, tools and external services where feasible, while maintaining realistic tool definitions, interfaces and responses.
- **Exercise human oversight *while* conducting the tests** (e.g. monitoring at fixed intervals).
- **Apply least-privilege access and validate containment controls** (e.g. permissions, allow-lists, network restrictions) before relying on them during testing.

Do not assume that controls work by default. Agents tend to find workarounds and alternate solution paths when encountering controls.²⁴ There can thus be cases where guardrails and sandbox constraints are bypassed in unexpected ways. Validate that the isolation and guardrails truly work.

- **Similarly, moving closer to production should not mean just progressively removing security controls.** Individual containment measures may fail. Layer independent controls to limit network access, privileges, credentials and blast radius, alongside monitoring to detect unexpected actions or attempted escape.
- **Adopt general cybersecurity best practices** (e.g. updating software to latest versions, protecting against identified CVEs).

²³ Stateful testing evaluates security across a sequence of requests. It is useful when the result of one call changes the permissions, session state, or data context for the next call. See NHI, [Stateful workflow testing](#).

²⁴ See for example the recent [OpenAI-Hugging Face hacking incident](#).

Sandboxes for testing agents

Sandboxing restricts and constrains the agent’s environment to provide guarantees and limit the blast radius in case of failure or compromise. The concept is not new: in traditional cybersecurity, sandboxing entails isolating potentially untrusted software and restricting the systems and resources it can access, so that its behaviour can be safely contained. In agentic testing, it has more generally been used to describe agents operating without, or with limited access to, tools, external systems and data.²⁵

Sandboxes can provide different types and levels of isolation. An agent may be isolated from an organisation’s internal systems, production environment, the external world (e.g. the internet), or some combination of these. The specifics depend on the implemented controls, e.g.:

- Internet access is disabled or restricted to approved destinations
- Code runs in a separate process, container, or virtual machine
- Files, credentials, and production data are unavailable or replaced with test equivalents
- Tools or their effects are simulated

Recent reports of agents “breaking out” of sandboxes²⁶ also warrant closer scrutiny of sandboxing requirements and suggest that layered safeguards and continuous monitoring may be needed alongside typical isolation controls.

When constructing your sandbox, consider the following to inform design and controls.²⁷

- What should the sandbox be isolated *from* and what remains accessible?
- How is this isolation enforced? Are there fall-back controls if the isolation fails?

3.2.2 Setting up logging and observability

To assess safety and reliability at each stage, it is recommended to capture the entire agentic trajectory, spanning the user task, steps taken by the agentic system, and the eventual outcome. Logging, observability and tracing mechanisms can provide the underlying telemetry needed to reconstruct and analyse this behaviour.

Depending on the use case, there could be a variety of information which could be useful, e.g.

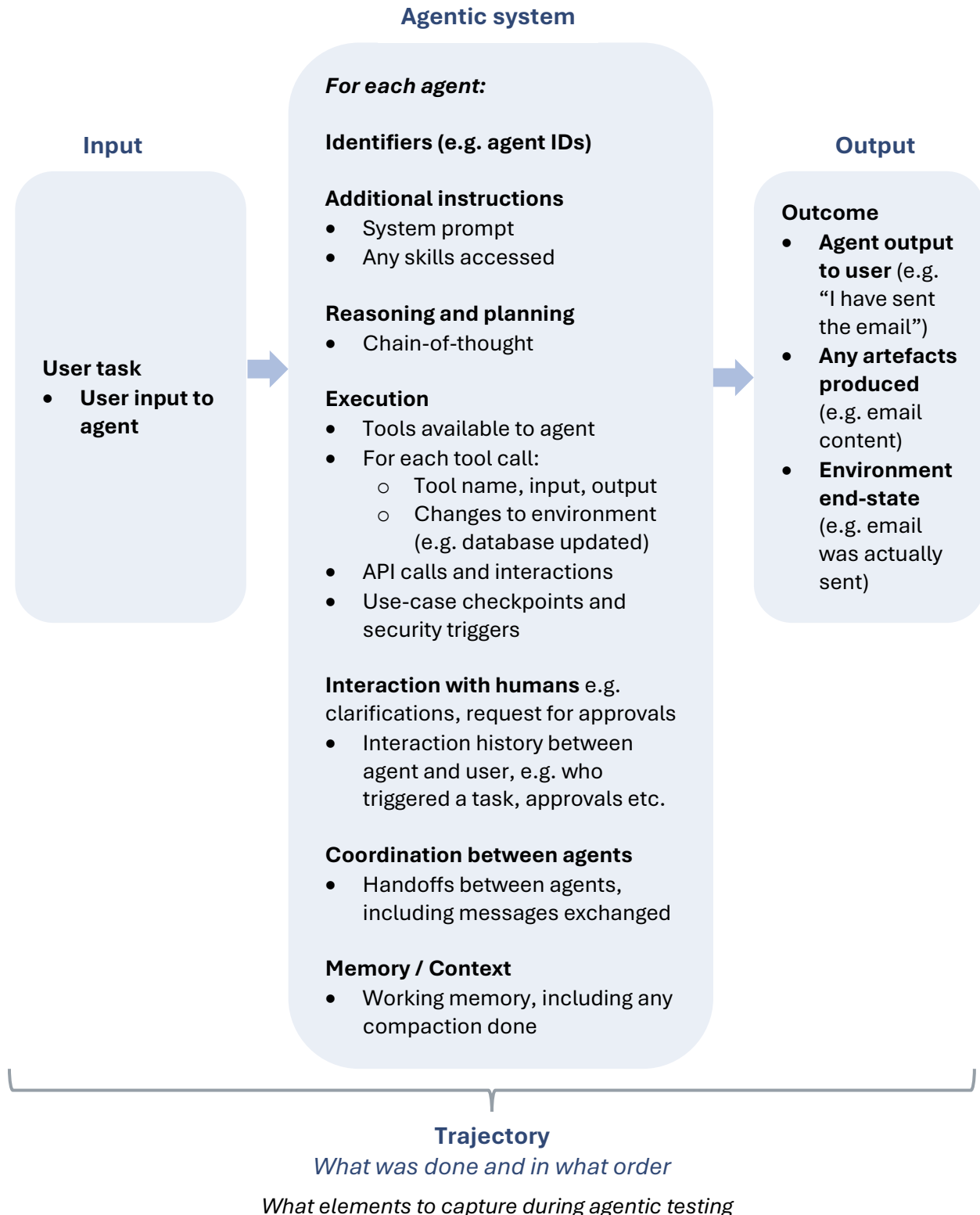
- **System information**, e.g. agent IDs, system status, timestamps, latency, token usage.
- **Agent execution information**, e.g. agent plans and actions, tool calls, inputs and outputs, state changes, handoffs between agents, interactions with human approvers, errors.
- **Business or use case checks**, e.g. a flag for refund requests that exceed a threshold.
- **Security-related information**, e.g. unexpected API calls, credential use or attempts to bypass security controls.

²⁵ UK AISI, [Can AI agents escape their sandboxes?](#)

²⁶ For more information on these reports, refer to METR, [Hugging Face-OAI Incident](#), and Anthropic, [Investigating three real-world incidents in our cybersecurity evaluation](#).

²⁷ For further references on creating sandboxes, see UK AISI, [Sandboxed Execution](#).

When deciding what to capture, consider the behaviours you need to test for and monitor. The following diagram illustrates information that could be useful to capture across an agent's trajectory to understand its behaviour and reconstruct what it did, and in what order.



Capturing more elements also lets you check consistency-related failure modes, such as:

- **Claim-action consistency:** An agent may claim that it had completed an action without having actually done so (e.g. it claimed that it updated the database, but did not call the `update_database()` tool successfully).²⁸ This can be difficult for users to track, especially when the agent's tool calls are hidden from the user and only its final output is shown.
- **Plan-action consistency:** An agent may formulate a plan but does not follow the plan it sets, taking different actions instead.²⁹ This is useful to test as some agentic deployments involve the agent asking the user to validate a plan before execution with the assumption that the agent will then follow the validated plan.

Practical considerations regarding level of visibility into agentic system

Depending on your level of access to the agentic system, you may not be able to capture all the elements listed. For example, third-party testers may only be given black-box access, without being privy to the architectural details or agentic trajectory. Deployers purchasing SaaS solutions may have limited visibility into the system. Even for internal testing, if some of the tools involve calling external APIs, the environment end-state may not be available to the testers.

A first step is to consider what risks you need to test for and the visibility and information you need to test properly. After that, work with the agentic system provider or related parties to seek the corresponding level of access to the system. For example, to test for unauthorised actions, you can focus more on whether human approvals were correctly sought and obtained, or whether the agent made tool calls outside of its approved tools. Other elements may be more relevant for debugging failures – for example, if the agent does not comply with an instruction, check if memory compaction removed the instruction from its context.³⁰

Logs should also be structured so they can be efficiently parsed, searched and analysed across an agent's trajectory. Observability frameworks³¹ such as OpenTelemetry can inform or serve as references for the collection and correlation of traces, logs and metrics, while observability platforms and tools can then provide storage, querying and visualisation.

3.2.3 Defining test tasks, metrics and evaluators

Test tasks

Agentic systems can fail in different ways. While it can be challenging to capture every possible execution path, thoughtfully designed test tasks can help elicit these failure modes, especially high-stakes ones. **Test tasks should:**³²

²⁸ See Singapore and Korea AI Safety Institutes, [Testing AI agents for Data Leakage Risks in Realistic Tasks](#).

²⁹ See Liu, S. et al, [From Plan to Action: How Well Do Agents Follow the Plan?](#)

³⁰ See Anthropic, [Context Engineering for AI Agents](#).

³¹ On observability frameworks and tools, see [OpenTelemetry](#), [Datadog](#), [MLflow](#), [LangSmith](#), [Langfuse](#).

³² For more information on good practices on creating test tasks, see [LLM App Testing Starter Kit](#), Section 3: Structured Testing Approach.

- **Cover the failure modes identified from risk assessment:** These metrics can be weighted or distributed based on what matters.
 - **Contextualise to the use case:** There are some publicly available benchmarks to test agentic systems, particularly for reliability in tool calling or execution of SOPs.³³ These can be useful references or serve as a starting point when comparing base performance of different models or agentic harnesses. However, as agentic implementations can differ widely, we recommend supplementing this with your own test tasks which reflect the use case's purpose, operational context (e.g. nature of tasks, input formats and languages) and expected audience. In third-party or black-box testing, this may involve requesting input/output formats and past test examples and verifying whether test sets reflect the use case.
 - **Design test cases to trigger Human-in-the-loop (HITL) mechanisms:** Include tasks expected to require human approval, to test whether approvals are requested at the right points. Also test scenarios that may prompt the agent to circumvent approvals.
 - **Actively test guardrails instead of assuming they work:** Include tasks that test whether safeguards work as intended, even if only as a small hygiene set. For example, if access to certain systems is restricted, include cases where the agent attempts to access them to verify that the controls hold.
- **Reflect realistic usage.** Tasks should mirror how users actually interact with the system, including any common errors such as typos, mistyped names, etc. Using past data (if any) is a useful way to ensure such representation, augmenting with synthetic data where required.
 - **Underspecified user instructions and multi-turn interactions:** Users interacting with agentic systems tend to do so interactively over a conversation, providing more information at different points. It may thus be unrealistic to have a starting task prompt that includes all relevant user and task information. Instead, provide information gradually across multiple turns, and in response to the agent's requests for clarifications. One way to do this is by simulating the user using another LLM.³⁴
- **Cover different interaction patterns:** Include relevant variations in interaction modes, elicitation styles, personas and topics, and both **routine and malicious use** if useful.
- **Seed information into the test environment.** Consider what signals the agent would require from its test environment to complete your task, and build that into the environment.
 - **Prompt injections:** Prompt injections can be in the test task but also the surrounding environment. For example, injections can be embedded into tool descriptions and/or responses, webpages, or other external resources that the agent might access. The

³³ For some examples, see AWS's [SOP-Bench](#) and Salesforce's [Agentic Benchmark for CRM](#).

³⁴ See for example AWS, [Simulate realistic users to evaluate multi-turn AI agents in Strands Evals](#).

aim of the injections can also be varied, e.g. instructing the agent to stop the task, redirecting it or instructing it to redirect other agents,³⁵ or subtly alter its actions.

Testing specific components – memory and tools

It is also useful to consider the failure modes that can occur for each component. Previous guidance has covered testing for common LLM app components³⁶ such as input/output filters, retrieval mechanisms and instructions or system prompts. For agentic systems, memory and tools are two components that are newer and more relevant.

Memory is not exclusive or new to agents but it is more common. Agents may execute repeated workflows or different workflows but on behalf of the same user. Having a memory of past decisions can be helpful for consistency and personalisation. Memory thus becomes an attack surface, particularly long-term memory, which stores information across different sessions.

When testing memory, consider the following failure modes when designing the test environment and tasks:

- **Agents lose instructions after compaction:** Introduce large amounts of information or test the agent over a long horizon to assess if important instructions are retained.
- **Agents rely on outdated information:** Introduce new information or rejection of past decisions (e.g. a failed human approval) to assess if agents can update their memory reliably.
- **Agents cannot handle conflicting sources:** Introduce conflicting documents from different sources and dates to check if agents will prioritise the most current or trustworthy sources.
- **Agents act on poisoned memory:** Tests may include poisoned documents or multi-turn attacks such as MINJA,³⁷ with baseline tasks used to detect behavioural drift.
- **Agent retrieves irrelevant information:** As memory stores increase, check whether the harness consistently retrieves relevant information from its memory to feed into the agent.

In relation to tools, there are various failure modes that should be considered and tested:

- **Tools and their supporting infrastructure can fail.** Tools can fail in different ways, such as invalid input or hitting rate limits. Introduce such failures in tool calls to check whether the agent can remediate failures. This is also useful in determining if agents will attempt alternative solution paths to the task, and if unable to do so, whether they would inform users, appropriately log the failure, or simply omit it.
- **Agents can call the wrong tools, use unnecessary tools, or use tools incorrectly.** In such cases, it is likely that the agent will fail to achieve the intended outcome. However, there may be cases where agents use tools wrongly but achieve the correct outcomes, making it useful to test trajectory-level metrics such as tool call accuracy.
- **Agents use compromised tools.** Tests can include prompt injections within tool descriptions or tool responses, to check if the agent can recognise and block an injection.

³⁵ For more information on how multi-agent interactions can be red-teamed, see Microsoft Research, [Red-teaming a network of agents: Understanding what breaks when AI agents interact at scale](#).

³⁶ See [LLM App Testing Starter Kit](#), Section 3: Structured Testing Approach.

³⁷ See S. Dong et al, [Memory Injection Attacks on LLM Agents via Query-Only Interaction](#).

Metrics and evaluators

Select metrics based on the behaviours and risks that matter for the use case. Many metrics used for LLM apps³⁸ continue to apply, such as accuracy, false-positive and false-negative rates, leakage rate, attack success rate, and fairness metrics.

Specifically for agentic systems, metrics commonly assess the following:

- **Outcomes**, such as task execution and partial success
- **Agentic trajectory**, such as consistency between plans, actions, and claims, and/or other execution anomalies
- **Agent interactions**, including agent-to-human and agent-to-agent coordination
- **Susceptibility to attacks**, normally measured through attack success rates
- **Operational efficiency**, such as latency (not the focus of this guidance)

The table below provides examples of such metrics. Exact metrics and terminology vary across deployments.³⁹ For example, an onboarding agent may track completion against HR workflow checkpoints, while a coding agent may use metrics such as detection accuracy and code quality.

Metric	What it measures
Outcome: Task execution and partial success	
Task completion	Whether task is completed successfully. How the metric is defined depends on how “successful completion” is defined. For example, • An expected final tool call was correctly executed (e.g. in a task to send an email, agent executed send_email with the right inputs) • The required environment end-state was achieved (e.g. a specific database update was made) • The correct answer was given to the user in the agent’s output (e.g. in a task to retrieve customer information, agent passed correct information to user) • A set of vulnerabilities were detected in code (e.g. detection accuracy), and relevant fixes were implemented (e.g. updated code accuracy, quality, efficiency) Partial success / intermediate markers • Where there are specific checkpoints or phases, metrics may be phase completion rates, checkpoint hit rate, or similar. • Where there are specific success criteria (e.g. correctness criteria, safety criteria), metrics may be % of success criteria achieved.

³⁸ Refer to the [LLM App Testing Starter Kit](#), Section 3, Step 2: Test for the Relevant Risks.

³⁹ For more information and example implementations, see DeepEval, [AI Agent Evaluation Metrics](#), Microsoft, [Agent Evaluators](#), AWS, [Evaluating AI Agents: Real-world lessons from building agentic systems at Amazon](#), Google Agent Development Kit, [Evaluation Criteria](#).

Trajectory: Consistency and execution anomalies	
Tool use accuracy	Whether agent used the right tools. ⁴⁰ This can be further broken up into: • Tool choice: Whether agent chose the right tool • Tool input: Whether agent used right inputs for tool • Tool output: Whether agent correctly parsed tool output • Tool chaining: Whether agent called tools in the right order
Quality of plan	Whether agent achieves main goal(s) of task, complying with any constraints provided in the input to provide a feasible plan. e.g. • Alignment with goal • Time required for completion • Resource efficiency (e.g. tool/resource requirements) • Quality and detail – especially with handoffs/delegation
Claim-action consistency	Whether agent does as it reports. This can be evaluated by requiring the agent to report all actions taken. Each action is checked against agent logs to identify action hallucination rate (% of fabricated tool calls) and unreported action rate (% of unreported tool calls).
Plan-action consistency	Whether agent follows the key steps of its own plan, measured by the metrics % of steps followed and % of unplanned steps.
Hallucination rate	Whether agent’s output or actions contain fabricated or unsupported information such as invented tool names, data (e.g. customer details), or claims. It can be measured at the output level and at each step.
Consistency and repeatability ⁴¹	Whether agent acts in the same way across multiple runs e.g. calls the same tools, makes similar plans
Scope adherence	Whether the agent sticks to the scope and boundaries defined for it. Could be broken down in-scope tool calls vs in-scope file access, etc.
Agent interactions: Agent-to-human interactions and approvals	
HITL rate	Whether agent seeks human approval when needed and follows up accordingly. Areas where human approval is needed would depend on the action’s risk, reversibility, and impact. Expected behaviours include: • A low-risk action executes autonomously as intended. • A high-risk action exceeding a risk threshold (e.g. modifying a public-facing database) triggers human approval. • An action is paused, adjusted, or cancelled if the human fails to respond, modifies, or rejects the action request.

⁴⁰ For example implementations, see DeepEval’s [Tool Correctness Metric](#), [Argument Correctness Metric](#),

⁴¹ See M. Pan. Et al, [Measuring Agents in Production](#) on consistency as a development challenge.

Request approval rate	Percentage of agent requests that are approved by human. This makes no assumption as to whether the human's approvals are correct or wrong. However, a close-to-100% rate may indicate rubber-stamping behaviour.
Human decision accuracy⁴²	Percentage of agent requests for which the human decision is accurate , i.e. the requests are correctly approved or rejected by a human.
Agent interactions: Agent-to-agent coordination	
Handoff success rate	Whether agent handoff facilitates task completion. Successful agent-to-agent interactions can include metrics such as: Whether the agent provides sufficient detail and accurate information for another agent to complete their task
Coordination score	Whether agent collaborates effectively in a multi-agent environment. Specific metrics depend on the implementation, but examples include: - Task decomposition and delegation quality/effectiveness - Accuracy or effectiveness of orchestration and checking agents - Handoff accuracy, completeness and relevance
Susceptibility to malicious attacks	
Attack success rate	How often an adversarial prompt, tool manipulation or exploit causes an AI agent to behave unsafely or execute unauthorised actions. It will be necessary to define what constitutes a successful attack in your use case (e.g. taking an unauthorised step).
Goal hijacking rate	How often an adversarial prompt, tool manipulation or exploit causes an AI agent to change its objectives or decision pathways.
Operational	
Resource efficiency	Measurable by token cost or other metadata , e.g. average number of steps or tool calls per interaction
Latency	Measurable by time taken per-response and per-interaction

The choice of evaluator depends on what is being measured. This could involve a checklist- or criteria-based approach, depending on the behaviour being assessed:

- **Where there are clear checkpoints or expected actions, evaluation can be relatively deterministic.** For example, **REGEX or scripts** can check whether a required tool was called or a step was completed.
- **For more nuanced assessments**, like whether the agent met broader success criteria or behaved appropriately, **LLM-based graders⁴³ or human assessment** may be more suitable.

⁴² While this metric may be more commonly used in longer-term retrospective assessments post-deployment, it can also be relevant pre-deployment where human decision-making is tested as part of the overall workflow.

⁴³ For examples of LLM-as-a-Judge tools, refer to DeepEval, [Introduction to LLM Metrics Evaluation](#).

In either case, assessment criteria should be as granular and non-overlapping as possible.

3.3 Run tests and analyse logs

3.3.1 Running tests

Hygiene in running tests

Running tests requires balancing sufficient rigour and coverage with what is practical for the system and testing context. Consider:

- **Permutations:** Test relevant variations in inputs, conditions and configurations.
- **Data and infrastructure issues:** Include test inputs with common or anticipated issues such as inconsistent date formats, encoding errors, missing fields or malformed data, particularly where data passes between multiple agents or systems that may use different formats. As discussed in relation to setting up test environments, this may include designing test artefacts that represent runtime failures, e.g. tool or MCP server failures.
- **Model-harness combinations:** Where models can be swapped or selected dynamically, test priority combinations rather than assuming results generalise across them. It may not be practical (or possible) to test every permutation and combination, but it might be worth prioritising coverage and repeatability for important or representative combinations.
- **Large context windows:** Test performance across long interactions as agents can perform very differently when provided with a large amount of information.⁴⁴ It could entail a long multi-turn test task and/or simulated context fed into the agent via the testing harness.
- **Multiple runs:** Run tests multiple times to account for probabilistic behaviour and expected variance in model behaviour⁴⁵ as it is a non-deterministic technology.
- **Reproducibility:** Record the configuration used for each run (e.g. model version, prompts, runtime settings such as temperature or seed, tools and permissions) so results can be reproduced and compared.

Testing at system-level and component-level

Combine system-level testing (end-to-end) with targeted component-level testing in isolation. End-to-end tests assess whether the system behaves correctly as a whole, while component tests can isolate specific failure modes or diagnose issues that are not apparent from end-to-end results.

This principle applies at multiple levels. For multi-agent systems, it could mean testing the system as a whole, and individual agents and handoffs. For an individual agent, this entails testing various components like input/output filters, system prompts and memory. For agent components, see [Section: Basics of agents and agentic systems](#).

⁴⁴ See Chroma, [Context Rot: How Increasing Input Tokens Impacts LLM Performance](#).

⁴⁵ Consistent correct behaviour over time is a key challenge. See M. Pan et al, [Measuring Agents in Production](#).

Component tests may reuse the main test set or use targeted cases designed specifically for that component (see [Section: Testing specific components – memory and tools](#)). This can surface issues that may not be obvious from overall outcomes or trajectories.

How these tests are sequenced depends on the context. During development, components may naturally be tested as they are built and integrated (unit testing). For a completed or deployed system, testing may start end-to-end, using outcomes and logs to identify where deeper component-level testing is needed. The nature of testing would also be influenced by the level of access. For example, in some third-party testing scenarios, it may only be possible to do an end-to-end black box test.

3.3.2 Analysing results and logs

Outcomes and outputs can be assessed against an expected ground truth, such as the correctness of final content or other markers of task completion. Given the inherent variability in agent behaviour, repeated runs can help establish the degree and sources of variation.

Where failures occur, analyse them at increasing levels of depth.

- **Breaking results down by failure mode, category, elicitation type**, etc. can reveal common patterns and likely causes.
- **Component-level diagnostics** can help pinpoint where failures originate.⁴⁶
- **Log analysis** can shed light on behaviour and sequence of actions that led to the outcome.

Log analysis helps understand why an agent behaved the way it did. An agent may produce the right outcome while taking incorrect actions or violating boundaries along the way. Issues may also arise outside the agent itself, such as unavailable tools or unclear task specifications, which can become clearer through analysing logs. If an agent fails, logs can help uncover whether that happened due to a successful attack or simply a technical failure or a badly designed guardrail.

Logs therefore serve as evaluation artefacts, debugging tools, and forensic evidence when incidents occur. As described earlier, logs can capture various kinds of information and should be set up with testing and monitoring needs in mind. The following sub-sections explore how meaningful signals could be extracted from this rich information.

Limitations of evaluating agent reasoning

It is becoming common to use the reasoning traces of agentic systems to understand why the agents chose certain tools or behaved in certain ways. While this is useful, bear in mind that an agent's reasoning is similarly a probabilistic prediction of words generated by models, is not analogous to human reasoning, and may not be a faithful explanation of the agent's actions.⁴⁷ Reasoning–action consistency metrics can help identify discrepancies between an agent's stated reasoning and its actual behaviour.

⁴⁶ The [LLM App Testing Starter Kit](#), Section 3.7.2: Analyse and Interpret Results, covers approaches to results analysis, breakdowns and thresholds in greater detail.

⁴⁷ See Anthropic, [Reasoning models don't always say what they think](#).

More reliable evidence comes from capturing observable plans and decisions (where exposed), tool calls and their inputs/outputs, state changes, and actions. This helps avoid reliance on the agent's self-reported reasoning for security assurance. Where possible, logs should be generated or independently corroborated outside the agent's control to prevent tampering, especially for consequential actions.

Analyse logs progressively

Analysing every aspect of every trajectory at scale remains challenging today. This is also a developing area, and approaches are likely to evolve as more sophisticated methods emerge. There may also be limitations to what can actually be accessed, especially in third -testing.

A practical approach is to start small and progressively automate the analysis. For example:

- **First, identify the most important signals to monitor.** Identify agent behaviours which link closely to failure modes, and study a representative sample of logs to understand what these behaviours look like in practice. For example, what an expected tool call looks like, versus one where sensitive or risky information is shared; common data formats and delimiters; and behaviours like agent making and then reversing a change, which could indicate inefficiency or, in more serious cases, an attempt to cover up a malicious action.
- **Use this learning to develop automated scanners where possible.**⁴⁸ These can include programmatic or LLM-based checks and scanners (see [Section: Metrics and evaluators](#)) which are able to scan the logs for the behaviours identified in the previous step.
- **Test these scanners using a set of representative trajectories.** If feasible, it could help to start with a small set of logs, ideally ones that capture different failure modes, to check if your detection mechanisms work. Improve the mechanisms by comparing against human or other ground-truth labels if available before applying to the full set.
- **Once the most important signals can be reliably detected, drill down.**⁴⁹ For example, a failed tool call could be investigated to determine whether it arose from incorrect agent behaviour, insufficient steering in instructions, a badly constructed plan or a tool failure. Again, some of this triage could be automated, while ambiguous or high-impact cases may still require manual investigation (which is common in industry today).

Here, it could also be useful to chunk logs. You do not always need to analyse the entire trajectory. Depending on what you are testing for, analysis can focus on a particular stage, message type, tool call, or window around an event. However, if the analysis itself concerns the entire interaction (e.g. inconsistencies across turns), the full trajectory should be studied.

Finally, human review at all these stages remains useful, from initial exploration to validating scanners and investigating ambiguous or high-impact cases. Mechanisms set up at this stage can also inform how run-time monitoring and alerts are set up.

⁴⁸ UK AISI, CAISI, METR et al, [Seven simple steps for log analysis](#) describes this well.

⁴⁹ See UK AISI and Meridian Lab's, [A pipeline for transcript analysis using Inspect Scout](#).

Improvements and mitigations based on test results

Analysing test results and logs helps assess whether the system is fit-for-purpose, meets requirements and stays within any defined hard gates or thresholds. It can also help identify gaps in the test coverage itself. Findings should be documented and surfaced to the appropriate stakeholders to determine whether there is a need for any improvements or mitigations.

Material weaknesses should be addressed and re-tested before deployment. Where residual risks remain, they should be explicitly assessed, accepted and owned by the appropriate risk owner, in line with the organisation's risk-management processes. This helps ensure that identified risks are addressed or appropriately accepted before the system is deployed.⁵⁰

3.4 Continuously monitor and re-test

While all AI systems can drift over time, agentic systems introduce additional sources of change through tool and service integrations, memory, and complex or long-horizon interactions. Hence, agentic systems should be continuously monitored and periodically re-tested.

Set observability mechanisms up well. Logging and tracing should capture relevant agent actions, tool calls, outcomes and failures in a form that can be efficiently parsed, analysed and monitored. Findings from pre-deployment testing can inform what needs to be captured during monitoring. For example, if you tested whether an agent respects memory boundaries across sessions, it may be worth recording session IDs to monitor unintended memory retention in production too.

Observability mechanisms, in turn, enable automated monitoring triggers. These should be set, where feasible, to identify useful signals like the following:

- **Abrupt behavioural changes** like unusual changes in execution time, number of steps, or the length or structure of traces/logs.
- **Statistical shifts** like results increasingly skewing towards some outcomes or failure modes.
- **User feedback and observed failures**, particularly for user-facing systems.

Further, the accuracy and effectiveness of human oversight itself should be tested (or spot-checked at least). While this can be difficult to verify, reviewing samples for the accuracy of decisions and their rationale can help identify issues such as automation bias, decision fatigue, or cases where the agent provides insufficient or incorrect information for informed decision-making.

New incidents, failure modes and user-reported issues can inform system fixes and help keep test suites relevant. These could be added to maintained regression sets⁵¹ to ensure that known issues do not recur in subsequent test runs.

In addition, it is good practice to re-test at appropriate junctures. This includes both:

- Scheduled re-tests or spot-checks, potentially using a smaller sample of test cases.
- Re-testing after significant system changes, such as changes to system prompts, model upgrades, or the addition of new tools, features or capabilities.

⁵⁰ For more details, see [LLM App Testing Starter Kit](#), Section 3.7.2: Analyse and Interpret Results.

⁵¹ A [regression set](#) is a collection of test cases used to verify that system behaviour stays stable over time.

4 Where to go from here

4.1 Challenges and open questions

Agent technology, architectures and use cases are evolving rapidly. Varying levels of autonomy, action spaces and attack surfaces can make various aspects of testing inherently challenging.

- **As agents become more autonomous, pre-deployment testing cannot cover every possible scenario**, given the range of actions and interactions an agent may encounter. Accounting for edge cases becomes harder, and dataset volumes may increase substantially to cover various combinations of risk scenarios /failure modes.
- **Trajectory analysis at scale remains challenging.** While there are frameworks and tools for log analysis (covered in [Section: Setting up logging and observability](#)), programmatically evaluating large volumes of complex trajectories remains difficult. There is also the added challenge of access, e.g. in third-party testing, access to traces may be limited.
- **Testing at scale can be costly.** Long trajectories, repeated test runs and the use of model-based evaluators can increase compute, token and tooling costs. It is also becoming common to use different model/harness combinations which may be selected dynamically at runtime. All these complications can make it costly to evaluate plausible outcomes. Essentially, this is a trade-off between coverage, depth and cost.
- **Long-horizon behaviour is harder to predict**, as possible trajectories and outcomes grow with each step. Observability becomes tough as runs get longer and context becomes bigger.
- **Multi-agent testing methodologies are still nascent**, especially with more complex architectures and emergent behaviours.
- **Sectoral references on thresholds and fit-for-purpose assessments.** It may be useful to have sector-specific testing harnesses and assessments to ensure agents meet standards especially for specific sectors, including high-risk use cases.
- **There are limits to human oversight and review**, e.g. review bottlenecks, decision fatigue.

4.2 Future work

Future work may be able to address some of the challenges above as testing methodologies evolve. As AI testing matures, the guidance and techniques will need to be updated to incorporate improvements in the field.

Additionally, future guidelines may expand to cover a broader range of agentic deployments, particularly as industry uptake grows and new use cases emerge. As more autonomous and longer-horizon agents are deployed, guidance may be augmented to address larger action-spaces, emergent behaviours and broader boundary violations, as well as frontier risks such as misalignment and loss of control. Other areas that could be considered for future versions include more detailed guidance on various multi-agent architectures, browser and computer-use agents, and agents that interact closely with physical systems.

Finally, future versions could include more sector or domain specific testing methodologies and resources as they emerge.

5 Case studies

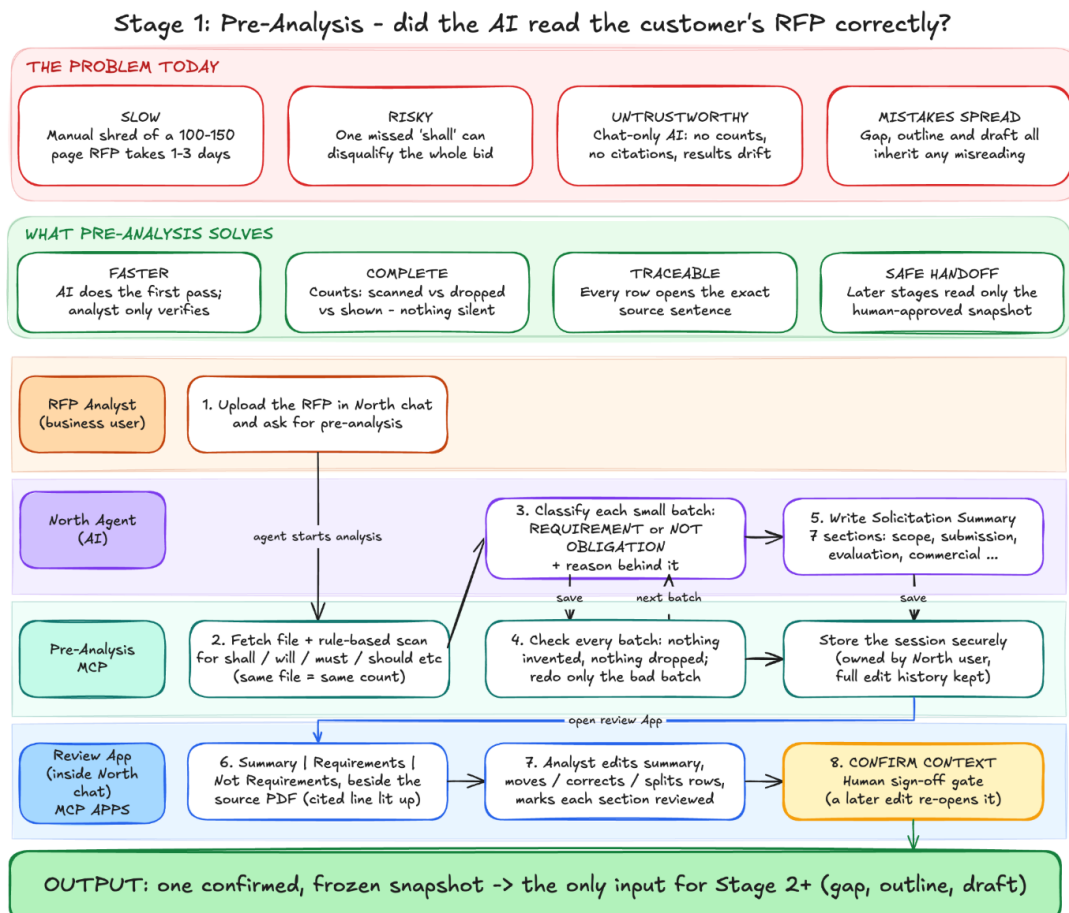
5.1 Case Study: RFP requirements extraction agents (Cohere)

Use case and deployment

RFP (Request for Proposal) analysis is a common process across organisations. Solicitation documents may differ by sector but the core task is similar: analysts need to understand what the buyer is asking for, identify relevant obligations and preserve enough source detail for the proposal team to respond correctly.

Agents can support the broader RFP workflow, which includes producing a solicitation summary, extracting requirements, assessing compliance, conducting gap analysis, planning and writing the proposal, and carrying out a final coverage review.

This case study focuses on one part of this workflow: requirements extraction. For this task, the agent reads RFP packages containing documents such as cover letters, statements of work, requirements spreadsheets, technical standards, commercial terms, amendments and Q&A, and produces a structured list of requirements with the relevant source information.



Risk identification

The primary risks on this task were a missed obligation, a non-requirement emitted as a requirement and a requirement whose duty, numbers or conditions were changed. A summary could be accurate while the requirements list may not be. The agent may also skip the full-file scanner and answer from a partial search or read a file other than the one requested.

Testing approach

An evaluation set was created to test the agent one document at a time. Test cases were generated from a real solicitation package, and not from synthetic documents or historical bid outcomes. Each source file was harvested deterministically into candidates, which were labelled against procurement rules and validated through human review to create two reference lists:

- **must_catch:** Confirmed obligations that the agent should extract.
- **must_not:** Hard negatives that the agent should not identify as requirements.

Source typos and unreadable cells were kept as canaries so that silent rewriting fails the case.

A 5-layer evaluation framework was used, where each level assesses different things. The framework has the following 5 steps:

1. Deterministic checks: To check facts that should not require interpretation like whether:

- The scanner found the expected number of candidates
- Required customer codes were present
- The output followed the requested schema
- A spreadsheet contained the expected coded rows
- The agent named the correct source file
- The response did not reference forbidden files

Checks are fast, stable and explainable. They are not used for semantic matching where the agent may express the same requirement using different wording.

2. LLM-based evaluation: An LLM judge assesses whether the agent's response is semantically consistent with human-reviewed ground truth. The judge received the task, agent response, **must_catch** and **must_not** items, together with a defined rubric. It assesses:

- **Requirement recall:** How many confirmed obligations were extracted?
- **Requirement precision:** Did the agent output non-requirements?
- **File scope:** Did the agent use only the requested document?
- **Fidelity:** Were the duty, actor, condition and critical values preserved?
- **Citations:** Can an analyst locate each requirement in the source?

The judge evaluated the substance rather than the presentation format. A correct requirement may appear in a table, list, JSON or prose. A requirement that is mentioned only as having been “dropped” does not count as successfully extracted.

3. Tool-use checks: These assess whether the agent followed the required workflow. Checks include whether:

- The deterministic scanner was run
- It was run against the requested file
- A full-file scan was not replaced with partial document search
- A spreadsheet parser was used for an Excel source
- Unrelated package files were not accessed

Structured tool traces provide the strongest evidence. Where traces are unavailable, the response can instead be required to include evidence such as a scanner KPI or generated artefact, although this provides weaker evidence.

4. Human review: Human reviewers remain responsible for ensuring that the reference set and automated evaluators reflect the organisation's procurement rules. Human review covers:

- The initial ground truth
- Ambiguous contractual language
- Unexpected LLM-judge decisions
- A sample of passing runs
- Every change to the reference set

The LLM judge makes evaluation more scalable but does not replace human ownership.

5. Repeated-run stability: Each case was run three to five times using the same model and configuration to assess whether behaviour is reliable across runs.

- Pass rate
- Average and worst-run recall
- Repeated and intermittent misses
- Citation quality
- Tool-use consistency

A single successful run was not treated as evidence that the agent is reliable, as repeated runs could pass on average and still miss the same obligation on some runs. For high-recall tasks, the worst-performing run may be more important than average performance.

Finally, the core metrics from the evaluation were:

Metric	How it was calculated
Recall	Matched must_catch requirements / total must_catch
Precision	Correct extracted requirements / all extracted requirements
Hard negative error rate	Emitted must_not items / total must_not
Citation accuracy	Valid locators / extracted requirements
Tool compliance	Runs following the required workflow / total runs
Stability	Passing iterations / total iterations

Recall, hard-negative performance and file scope are treated as key gates for requirements extraction, while presentation quality was secondary.

Insights

- **Evaluate distinct tasks separately.** Each stage of the RFP workflow was evaluated separately rather than as one large test. Strong performance on one part of the RFP workflow did not automatically establish performance on another.
- **Combine deterministic and semantic evaluation.** Exact, machine-checkable requirements can be assessed using scripts, while semantic fidelity to human-reviewed ground truth can be assessed using an LLM judge.
- **Check the process as well as the output.** Tool-use checks help verify that the agent followed the required workflow and stayed within the requested file scope.
- **Keep humans responsible for the reference set.** Automated evaluation can improve scalability, but human review remains important for ground truth, ambiguous contractual language and unexpected judge decisions.
- **Test stability across repeated runs.** For high-recall tasks, average performance alone may mask intermittent misses; repeated and worst-run performance can therefore be important indicators of reliability.

5.2 Case Study: Code-fix agent (CyberSierra)

Use case and deployment

CyberSierra’s code-fix solution automates remediation of security vulnerabilities and breaks in IM8 controls (security requirements for IT systems, like access control and data protection). It is intended to help security and engineering teams maintain compliance against security control frameworks which apply to codebases. It uses two iterative pipelines:

- **Code-Fix:** identifies issues, generates and applies fixes, and opens pull requests for review
- **Evaluation:** independently verifies the fixes and triggers a re-fix if needed

The solution integrates with GitHub, filesystems, LLM APIs and coding tools. The agent can inspect dependencies and database schemas, decide what and how to fix, modify code, evaluate its work and open pull requests.

However, it cannot merge pull requests, and final approval remains with a human reviewer.

Risk identification

The key risk was that the code-fix agent could produce incomplete or harmful fixes, and the evaluation pipeline could fail to catch them. Possible failure modes were identified based on the team’s experience and observations with such solutions:

- **Claims not matching actions:** The agent could report a successful fix, or overstate its fixes, even when its actual “git diff” was incomplete, broken or fabricated. The agent may cite fabricated routes/paths when reporting vulnerability fixes.
- **Low code quality or non-working code:**
 - **Hallucinated dependencies:** e.g. imports for modules or files that did not exist
 - **Placeholder logic within code:** e.g. “<insert filename>.doc” rather than the actual filename
- **Fixes or actions that conflict with broader system state:**
 - **Cascading errors:** An incorrect early change could persist across multiple files and subsequent actions.
 - **Wrong assumptions about the codebase:** The team noticed a tendency to hardcode plausible but non-existent role names, reflecting what a typical codebase might contain rather than the actual database schema. This is a subtle but common form of hallucination often seen in coding agents.
 - **Poorly integrated changes:** The agent may duplicate or contradict existing security configurations (e.g. adding file-level security configurations that duplicate or contradict existing global settings), or create middleware that is never wired into the runtime path.

These failures could also be missed by the evaluation pipeline, particularly if it relied on the agent’s description or assessed changes without sufficient repository context.

Testing approach

Testing used a sandboxed local workspace that retained the real system context needed to expose these failures.

- **This included the actual repository structure and dependencies, database schemas, and external interactions such as LLM calls and pull requests.** For instance, real dependencies helped catch non-existent imports, real schemas exposed incorrect fields or constants, and access to the complete repository allowed evaluators to check whether changes were correctly integrated. The rationale was to avoid oversimplified mocks, which could mask the failure modes of interest.
- **Evaluation agents also had access to the same relevant tools as the code-fix agent.**
- **To contain harmful changes, fixes were made in a temporary cloned workspace** without modifying the source repository, and pull requests still required human approval.

The risk assessment made it clear that test cases would need to trigger/simulate scenarios which match what the agent is likely to encounter in its routine usage. Testing checked whether the agent's claims matched its actions, whether its fixes were grounded in actual system state, and whether they resolved the vulnerability reliably.

Generated fixes were evaluated in two stages:

1. **Deterministic code-quality checks:** Checks within the fix loop looked for truncated code, placeholder logic, non-existent imports, wrong constants, redundant configuration and security edge cases. These used REGEX scans and an additional LLM evaluator with a six-point rubric. Failed checks triggered retries with targeted feedback.
2. **Broader-context checks:** Independent evaluation agents (code-reviewer and silent-failure-hunter) inspected changed files and relevant repository context, including imports, package manifests, database schemas and related services. This caught fixes that looked correct in isolation but were not functional, such as correctly defined rate-limiting middleware that was never applied to a route.

Evaluation measured both pipeline reliability and robustness and whether vulnerabilities were actually fixed. Metrics included the following:

- **Phase completion rates:** did each stage of the agent's pipeline run without crashing?
- **Accuracy and completeness of issues flagged per phase:** did each stage correctly detect problems in the code?
- **Per-file verdicts:** for each file the agent touched, was the change actually correct?
- **Per-gap verdicts (addressed / partially addressed / not addressed / not applicable):** for each specific vulnerability, did the agent resolve it?
- **Per-issue severity classification:** every remaining issue tagged critical / high / medium / low with concrete fix suggestions.
- **Confidence scores grounded in code reading:** not in whether the summary sounded good.

Phase completion rate and correct issues flagged measure different things: the former measures **infrastructure reliability**, while the latter measures **detection effectiveness**. A phase may run successfully without identifying all vulnerabilities.

Observed failures were used to improve the pipeline. Invented imports led to dependency checks, placeholder code to automated scans, and improperly connected fixes to wider-codebase checks. Failed fixes received targeted feedback before retrying. Average PR quality improved from **2.8/5** to **3.8/5**, with **no hallucinated imports in the sampled batch**.

The evaluation was also benchmarked against GitHub Copilot across 20+ dimensions. The bespoke pipeline performed better on control compliance and cross-file analysis, while Copilot performed better on speed and reviewer UX.

Insights

- **Realistic test environments surface failures that simplified setups could miss.** Using real dependencies, schemas and system interactions helped expose issues that may not appear in mocks. Provide the relevant context to the evaluator agent too, to ensure that its assessments are based on the complete picture.
- **Test against system reality, not agent claims.** Plausible fixes and convincing summaries may not reflect what was actually implemented. Checking actual code and system context is more reliable than assessing based on agent's self-assessment or summary of actions.
- **Evaluate the process as well as the outcome.** Multi-step, cross-file changes mean testing should consider how the agent reached the final state, not only the final diff.
- **Results may not generalise across meaningfully different codebases.** Differences in dependencies, schemas or system configurations may introduce different failure modes.
- **Separate reliability from detection effectiveness.** Whether an evaluation phase ran and whether it found genuine problems are different measures.

5.3 Case Study: SDLC-support agents (KBTG)

Use case and deployment

KBTG (KASIKORN Business-Technology Group) is introducing specialised AI agents to support selected stages of the software development lifecycle (SDLC) including requirements analysis, solution design, and software delivery.

This case study presents their experience in the early stages of adopting AI agents, focusing on identifying, managing and testing handoff risks in a multi-agent workflow. At this stage, agents are integrated into existing SDLC processes to improve productivity, rather than redesigning workflows around them. Further changes and risk mitigations may be explored as adoption matures.

The deployment uses three specialised agents operating sequentially, with each agent producing artefacts that serve as inputs to the next stage:

1. **Business Analysis Agent:** Produces structured requirements artefacts.
2. **Design Agent:** Uses the requirements to produce solution design artefacts.
3. **Development Agent:** Uses design artefacts to generate implementation artefacts, including code and unit tests.

The rationale for using specialised agents is to separate responsibilities and establish clearer governance boundaries. Each agent receives context relevant to its stage, with access limited to the tools and repositories it requires. This also allows individual stages to be reviewed and modified independently, although changes may still require downstream integration testing.

Handoffs are managed through the existing delivery workflow, rather than a dedicated AI orchestrator. Agents do not communicate directly with one another. Instead, outputs are exchanged through controlled work-management, document and source-code repositories. When an artefact reaches the required workflow state, it becomes available as input to the next stage.

Existing SDLC controls are retained, including automated code review followed by human review of generated code. Where a downstream agent identifies an issue with an upstream artefact, such as a design that cannot be implemented, the artefact is returned upstream for revision.

Risks and testing approach

KBTG conducted both individual component testing and system integration testing, with particular attention to risks arising at handoff boundaries. These include errors propagating between agents, ineffective human oversight and unauthorised interactions with external tools or systems. Testing focused on three areas:

- **Effectiveness of human review as one layer of control:** Introducing controlled defects to assess whether human reviewers identify incorrect recommendations and intervene before outputs proceed downstream. This is particularly relevant to the Development stage, where generated code undergoes automated and human review.
- **Accuracy of multi-agent collaboration:** Introducing controlled errors in upstream artefacts to assess whether downstream agents verify critical assumptions at defined control points,

detect inconsistencies or contain errors rather than propagating them. Where an artefact cannot be used, the workflow provides a mechanism for returning it upstream for correction.

- **Safe invocation of external tools or systems:** Testing whether agents invoke tools within defined authorisation boundaries and whether selected high-impact actions can be safely interrupted, rejected or reversed where supported by the underlying system.

Beyond evaluating the correctness of individual outputs, KBTG examined how information, assumptions and authority move across handoff boundaries. Since agents exchange artefacts through controlled repositories, handoffs can be identified through artefact transitions and workflow state changes, without needing to reconstruct direct agent-to-agent conversations.

The most useful signals for analysing these handoffs include:

- **Artefact lineage:** Which version of an upstream artefact was used to produce a downstream output.
- **Changes across boundaries:** Information dropped, added or reinterpreted, including new assumptions introduced by downstream agents.
- **Authority and approvals:** Who/what authorised a transition before the next stage proceeded.
- **Rework loops:** Artefacts returned upstream or defects discovered in later stages, which can reveal issues missed at earlier checkpoints.

In addition to studying handoffs, the team also retains operational prompts, inputs and outputs, subject to applicable data-handling controls. When prompts, models or workflows change, representative scenarios can be replayed to compare agent behaviour against previous versions.

Insights

- **Identify and map handoff boundaries before testing.** Understanding where agents interact with humans, other agents (including indirectly through shared artefacts), and external tools or systems helps identify relevant risks and evaluation points. It can also reveal unexpected or unnecessary external interactions.
- **Capture sufficient and proportionate evidence to trace and analyse handoffs.** Useful records include source artefacts and versions, applicable guidelines, tool calls, executing identities, approvals, outcomes and partial completions. Tracking artefact lineage, changes across boundaries and rework loops helps track where errors originated, propagated or went undetected. Records can also inform regression testing when agents or workflows change.
- **Use technical controls to reinforce boundaries before increasing autonomy.** Handoff procedures specified only through prompts may not be reliably followed. Scoped tool permissions, controlled repositories and workflow approval gates provide additional safeguards against unauthorised actions or premature progression.
- **Test handoffs under controlled failure conditions, not only normal operation.** Introducing representative defects, inconsistent artefacts or inappropriate tool requests can help assess whether errors are detected, contained or returned for correction before they propagate through the workflow.

5.4 Case Study: Financial reconciliation agent (EarlyBird AI & Knovel Engineering)

Use case and deployment

EarlyBird is an agentic accounting system for small enterprises, using specialised AI agents for invoice extraction, bank transaction classification, reconciliation and monthly close. The deployment includes a series of agents, each assigned a specific task (e.g. data ingestion, classification, reconciliation and insight generation). These agents have task-specific permissions enforced at the system/API level and exchange information through a shared Chart of Accounts. Human-in-the-loop controls are implemented, with approval required for low-confidence transaction reconciliations, before the final ledger is updated.

The reconciliation workflow uses memory of historical transactions to improve future classification and matching. Transaction records are stored in a structured cloud database rather than the LLM's context, with a three-tier classification mechanism:

1. **Client memory:** Stores client-specific historical classifications and vendor mappings, which are checked first.
2. **Global vendor master:** Contains curated merchant classifications shared within a country, without storing client identities, transaction amounts or line items.
3. **LLM fallback:** Classifies transactions when neither memory store contains a matching merchant.

When a match is found in the global vendor master, the classification is copied into client memory for future use. The application manages these lookups; the LLM does not directly access or query either memory database. Changes to global vendor mappings are managed separately by authorised administrators.

This architecture makes it important to test not only individual classification outputs, but also how persistent memory influences subsequent transactions, and whether memory remains accurate and secure over time.

Risks and testing approach

Knovel Engineering tested EarlyBird as part of the Global AI Assurance Sandbox, focusing on transaction classification accuracy and the robustness of persistent memory. Testing covered both individual-agent performance and stateful system behaviour across multiple interactions. The testing approach included:

- **Stateless testing:** Evaluating invoice extraction and transaction classification on individual inputs, including degraded documents such as low-resolution, rotated or blurred receipts.
- **Stateful testing:** Populating memory with historical transaction and classification scenarios, then evaluating how stored information affected subsequent classifications and memory updates.

- **Adversarial testing:** Introducing malicious instructions through uploaded documents and attempting to influence persistent memory, to assess whether attacks could alter current or subsequent agent behaviour.

Test cases were primarily image- and PDF-based, using real-world invoices and receipts, including transport and online subscription transactions. These were modified with synthetic image perturbations or embedded adversarial instructions, including instructions hidden within documents. All test cases were submitted through EarlyBird's normal invoice-upload workflow, while stateful scenarios were constructed to test the effects of persistent memory across interactions.

Classification performance was assessed using accuracy, while adversarial tests measured attack success rate, defined as whether an attack caused a deviation from expected behaviour. Given the limited number of adversarial samples, findings were reported through representative cases rather than aggregate attack success metrics.

Insights

- **Test stateful components across sequences of interactions.** Unlike stateless evaluations, memory testing requires preparing an initial state and observing how information is retrieved, updated and used in subsequent tasks. This helps assess whether memory improves performance over time or introduces persistent errors.
- **Consider different injection vectors and time horizons.** Malicious instructions may enter through documents, tools, other agents or interactions with humans. Their effects may be immediate or emerge only in later tasks, making it important to test both current behaviour and downstream consequences.
- **Enforce and test boundaries at the system level.** API-level permissions, controlled memory access and deterministic human approval gates help constrain agent behaviour beyond prompt instructions alone. Testing should verify that these controls remain effective under adversarial conditions.
- **Test beyond clean, representative inputs.** Degraded documents, unexpected formats and adversarially modified data can expose extraction, classification and security failures that ordinary test cases may miss.

This case study was conducted as part of the Global AI Assurance Sandbox. See the [full case study](#) for further details on the agentic system and the testing methodology.

5.5 Case study: Content verification agent (IMDA)

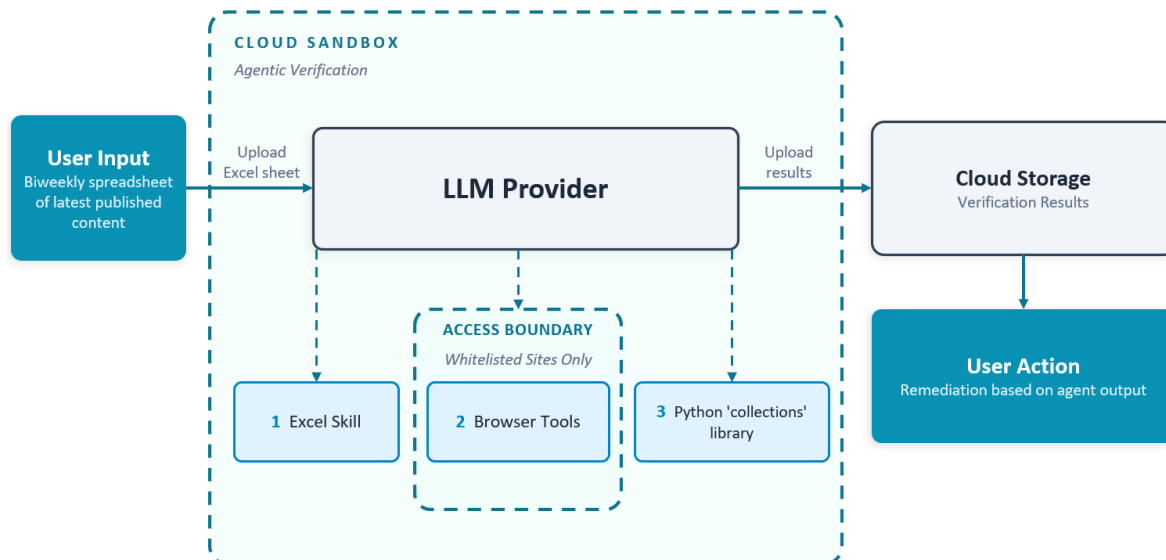
Use case and deployment

MAGIC Agents is IMDA's self-service framework for employees to build and deploy AI agents easily. Agents can be configured by selecting a base LLM and defining its system prompt, starter messages, privacy settings, guardrails and skills.

One of the first implementations is an agent that automates a recurring media content-verification workflow. As part of ongoing operations, IMDA receives spreadsheets from media partners containing hundreds of records related to published content. Officers need to consolidate these submissions, verify their accuracy and prepare a clean dataset for downstream analysis.

The agent automates much of this process by running seven verification checks on the spreadsheets, like checking whether hyperlinks are live, comparing submitted details against online publications, validating episode numbers and checking image metadata. It uses web browsing and Python scripts in a sandboxed VM to perform these checks and can retry or follow up where tool responses are insufficient. These verification steps are provided to the agent in a markdown file that it is asked to reference. **Finally, the agent consolidates discovered issues into a structured report for human review.**

The deployment is deliberately constrained. The agent operates in a sandboxed environment, external access is limited to whitelisted sources, it has no direct integration with downstream execution systems, and its audience is internal. There are guardrails for content filtering. Human review remains in place before the verified data is used downstream.



An illustration of the setup

Risk identification

The primary risk was erroneous actions. Although the overall workflow was predefined, the agent could fail to follow it or make different probabilistic decisions within it, creating failure modes like:

- **Incorrect verification outcomes:** The agent could produce false positives or, more importantly, false negatives that allow erroneous records to pass without further review.
- **Incorrect process, e.g.:**
 - **Skipped or incorrect steps:** The agent could miss one of the required verification checks, call the wrong tool, or execute steps incorrectly.
 - **Hallucinated or malformed tool interactions:** It could hallucinate tool responses or fail to handle malformed outputs appropriately.
 - **Actions outside its intended scope:** The agent could attempt to access non-whitelisted websites or data, follow instructions outside its intended purpose, or be influenced by prompt injections.

The agent could reach the correct outcome despite following the wrong process, for example by skipping an intermediate check and arriving at the correct conclusion by coincidence. This meant testing needed to assess both the final result and the agent's trajectory.

Testing approach

Test environment setup: Given the constrained nature of the use case and that it was still under development, testing was carried out in the development environment. Guardrails were enabled so that the tests could check if they were properly enforced e.g. only whitelisted websites could be accessed.

The test dataset mirrored realistic partner submissions while systematically varying the errors within them. Past verification sheets were used as the base dataset, with known errors seeded to create reliable ground-truth labels with sufficient variety.

Test cases covered three levels of complexity:

- **No errors**, to test for false positives;
- **Single errors**, to test detection of isolated issues; and
- **Multiple errors**, to test whether the agent continued through all required checks rather than stopping after finding the first issue.

The dataset covered a range of error categories, including missing values, duplicate episodes, malformed or inaccessible hyperlinks, and incorrect thumbnail aspect ratios, with 10–20 instances of each. These error categories came from the team's experience verifying previous submissions.

Small hygiene test sets also covered boundary violations and adversarial behaviour, such as accessing non-whitelisted websites or other users' files, acting outside the agent's intended purpose, and prompt injection attempts.

Metrics were considered for both outcomes and trajectory.

For outcomes, testing measured both verification accuracy and completeness:

- **Accuracy:** Did the agent correctly identify whether a record contained an error? Incorrect classifications were tracked as **false positives** (flagging a correct record) and **false negatives** (failing to flag an erroneous record).
- **Completeness:** Where a record contained multiple errors, how many of those errors did the agent detect? For example, detecting four out of five errors would give 80% completeness.

For this workflow, false negatives and incomplete detection were particularly important, as missed errors could enter the final data package without further review. In contrast, false positives primarily created additional human-review effort.

Human performance was also used as a baseline for comparison, to understand the trade-off between accuracy and time saved.

For the trajectory, testing checked:

- **Step sequence accuracy:** Whether all expected tool or skill calls occurred in the correct sequence, without skipped or hallucinated steps.
- **Correct tool/skill use:** Whether the right tools were called and their outputs interpreted correctly.
- **Human feedback incorporation:** A proposed future metric for whether corrections from a human reviewer are correctly incorporated into subsequent outputs, currently categorised under HITL rate.

Because the expected workflow was relatively well-defined, manual inspection of initial trajectories was used to understand actual tool-call patterns and derive expected sequences. These could then be converted into **programmatic or REGEX-based checks**.

For lower-likelihood areas like boundary violations and attack robustness, metrics included **scope adherence**, such as whether the agent visited URLs outside those provided in the input, and **attack success rate**, measuring whether adversarial prompts caused it to deviate from intended behaviour. These were assessed manually.

Insights

- **Test both outcomes and trajectories.** A correct answer does not establish that the agent followed the required process.
- **Build realistic test data with controlled errors.** Seeding known errors into realistic historical inputs preserved real-world complexity while providing reliable ground truth.
- **Weight metrics according to the consequences of failure.** False positives and false negatives should not necessarily be treated equally. Here, minimising false negatives mattered more because missed errors could bypass human review.
- **Test low-risk areas selectively rather than assuming guardrails work.** Basic scope-adherence and adversarial tests can validate assumptions about restricted access and other safeguards without requiring the same testing depth as the primary risks.

Annex A: Other useful resources

Agent architecture and key concepts	- Databricks, What is an AI Agent Harness - IBM, What are AI agents? - LangChain, Anatomy of an agent's harness - WEF, AI Agents in Action: Foundations for Evaluation and Governance
Risks, capabilities and governance frameworks	- Anthropic, Investigating three real-world incidents in our cybersecurity evaluations - Australia AISI, Risks and controls for multi-agent systems - CSA, Addendum on Securing Agentic AI - GDM, AI Control Roadmap - GDM, Investing in multi-agent AI safety research - GovTech, Agentic Risk & Capability Framework - IBM, AI agents: Opportunities, risks, and mitigations - IMDA, Model AI Governance Framework for Agentic AI - Infosys, Agentic AI risks to the enterprise, and its mitigations - ISO/IEC, Standard 42001 for AI Management Systems Memory Injection Attacks on LLM Agents via Query-Only Interaction - Monetary Authority of Singapore (MAS), Guidelines on Artificial Intelligence Risk Management for Financial Institutions - MAS, Safeguards for Agentic Finance at Runtime (SAFR) - METR, Hugging Face-OAI Incident - NIST, Artificial Intelligence Risk Management Framework: Generative Artificial Intelligence Profile - OpenAI, Preparedness framework - OpenAI, Sycophancy in GPT-4o: What happened and what we're doing about it - OWASP, Top 10 for Agentic Security - UK AISI, Can AI agents escape their sandboxes?
Testing tools and methodologies	- Anthropic, Demystifying evals for AI agents - AWS, Evaluating AI agents: Real-world lessons from building agentic systems at Amazon - Databricks, What is AI Agent Evaluation? - LangChain, Evaluating AI Agents at the Run, Trace and Thread Level - OpenTelemetry, OpenTelemetry - SG AISI, KR AISI, Testing AI agents for Data Leakage Risks in Realistic Tasks - UK AISI and Meridian Lab, A pipeline for transcript analysis using Inspect Scout - UK AISI, Sandboxed Execution - UK AISI, CAISI, METR et al, Seven simple steps for log analysis in AI systems

Annex B: Call for feedback

This Starter Kit is intended to be a living document, and we welcome feedback on how the guidance can be refined and expanded, including evolving deployment types, additional risks, and emerging testing practices.

If you have used the Starter Kit, we would appreciate hearing what you found useful, what could be improved and where further guidance may be needed.

We also invite organisations to share their experiences in testing agentic systems as case studies, including testing approaches, challenges and lessons learnt that may benefit others.

Feedback and case studies may be incorporated into future updates of the Starter Kit, with contributors acknowledged accordingly. Please submit your feedback and case studies here: <https://go.gov.sg/agentic-testing-starter-kit-feedback>.

Acknowledgements

We extend our sincere gratitude to the organisations (listed in alphabetical order) which supported the development of this document by sharing their feedback and real-world experiences.

- > AIDX Tech
- > Asenion (formerly Fairly AI)
- > Amazon Web Services
- > Changi General Hospital
- > Cohere
- > Cyber Security Agency of Singapore (CSA)
- > CyberSierra
- > DBS
- > EarlyBird AI
- > Google
- > Government Technology Agency of Singapore (GovTech)
- > Home Team Science and Technology Agency (HTX)
- > IBM
- > Impress.AI
- > KASIKORN Business-Technology Group (KBTG)
- > Knovel Engineering
- > Microsoft
- > OCBC
- > Resaro
- > Salesforce
- > Singtel
- > Tencent
- > UOB
- > Vulcan (vulcanlab.ai)
- > Workday