Build reliable agent applications with the GitHub Copilot SDK

Own the lifecycle, policy, run records, and failure path

Session M6-01 | Module 6 | Advanced

Agenda

Segment	Time
The application reliability boundary	8 min
Client, session, and streaming lifecycle	12 min
Tools, permissions, and hooks	17 min
Tests and traces	10 min
Prepared demo and lab handoff	13 min

The application owns the control path

The SDK provides an agent session. Your application still owns:

- startup and shutdown;
- the data boundary;
- which tools exist;
- which calls may run;
- what the user sees when work fails;
- the records kept after the session ends.

A useful answer with an unsafe control path is still a broken application.

The scenario is deliberately small

```
Synthetic change record
        ↓
TypeScript CLI
        ↓
Copilot session
        ↓
Change request lookup tool
        ↓
ready or needs-review
```

The data is fixed. You build the controls around the model.

Manage startup and cleanup

```
import { CopilotClient } from "@github/copilot-sdk";

const client = new CopilotClient({
  telemetry: { otlpEndpoint: "http://localhost:4318" },
});

await client.start();
let session;

try {
  session = await client.createSession(config);
  await session.sendAndWait({ prompt });
} finally {
  await session?.disconnect();
  await client.stop();
}
```

Cleanup belongs in `finally`. A successful response is not proof that resources were released.

One client can own several sessions

Object	Owns	Close when
`CopilotClient`	Runtime connection and client options	The application exits
Session	Conversation state, tools, hooks, and events	The task is complete
Event subscription	UI updates from that session	The session is no longer observed
Collector connection	Trace export path	The local trace stack stops

Keep session state scoped by session ID. Remove it in the session-end path.

Streaming improves feedback and creates edge cases

```
session.on("assistant.message_delta", (event) => {
  if (!event.agentId) {
    process.stdout.write(event.data.deltaContent);
  }
});

session.on("session.idle", () => {
  process.stdout.write("\n");
});
```

The final `assistant.message` still arrives. Decide whether the CLI renders deltas, the final message, or both in different places. Printing both produces duplicates.

Events do not decide readiness

Session events can show:

- text deltas;
- tool requests and results;
- permission requests;
- session state changes;
- usage records.

The readiness verdict must still come from the application's deterministic data and rules. Do not infer readiness from a fluent answer.

A custom tool limits what the agent can do

```
import { defineTool } from "@github/copilot-sdk";

const lookupTool = defineTool("lookup_change_request", {
  description: "Read one synthetic change and calculate readiness gaps",
  parameters: {
    type: "object",
    properties: { requestId: { type: "string" } },
    required: ["requestId"],
    additionalProperties: false,
  },
  handler: async (args) => lookupChangeRequest(changes, args),
});
```

The schema narrows the input. The handler still validates the request ID before it reads the fixture.

Tool availability and permission answer different questions

Control	Question
Registered tools	Can the model choose this capability?
`onPermissionRequest`	May this requested execution proceed?
`onPreToolUse`	Does application policy allow these exact arguments now?
Tool handler	Are the input and data valid at execution time?

Use all four where the risk justifies them. One broad approval does not replace argument-level policy.

Deny the action before the handler runs

```
hooks: {
  onPreToolUse: async (input) => {
    if (
      input.toolName === "lookup_change_request" &&
      input.toolArgs?.requestId?.startsWith("RESTRICTED-")
    ) {
      return {
        permissionDecision: "deny",
        permissionDecisionReason:
          "Restricted change requests cannot be read by this assistant.",
      };
    }

    return { permissionDecision: "allow" };
  },
}
```

The denial reason should appear in the CLI and trace. Silent denial is hard to debug and easy to misread.

Keep permission handling narrow

```
onPermissionRequest: async (request) => {
  if ("managedApprovalRequired" in request &&
      request.managedApprovalRequired === true) {
    return { kind: "no-result" };
  }

  if ("toolName" in request &&
      request.toolName === "lookup_change_request") {
    return { kind: "approve-once" };
  }

  return {
    kind: "reject",
    feedback: "Only the synthetic readiness tool is allowed.",
  };
},
```

Deny unknown request kinds. Review new capabilities before approving them so they do not inherit permission by accident.

Hooks record and control the lifecycle

```
Session starts
  → request streams
  → tool call reaches pre-tool policy
  → permission decision
  → handler succeeds or fails
  → failure or error hook records the outcome
  → session ends and state is cleared
```

Hooks run inline. Keep them fast and deterministic.

Failure and error are different paths

Path	Example	Application response
Denied call	Restricted synthetic ID violates policy	Show the reason, record denial
Application result	Malformed or missing synthetic ID	Return a fixed result without invention
Tool execution failure	Handler throws or returns a failed execution	Record the failed tool event
SDK error	Runtime or model call fails	Notify the user, retry only when safe
Session end	Complete, abort, timeout, or error	Flush state and record reason

Do not turn every failure into a retry. Some failures are policy decisions.

Use the failed-tool hook for agent guidance

```
onPostToolUseFailure: async (input) => {
  audit.failedTools.push({
    toolName: input.toolName,
    result: input.toolResult,
  });

  return {
    additionalContext:
      "Do not invent a change record. Report the failed tool result.",
  };
},
```

This hook sees tool results marked as failures. It does not replace `onErrorOccurred`.

Use the error hook for SDK recovery policy

```
onErrorOccurred: async (input) => {
  audit.errors.push(input.error);

  return {
    errorHandling: input.recoverable ? "retry" : "abort",
    retryCount: input.recoverable ? 1 : 0,
    userNotification: "The readiness session could not continue.",
  };
},
```

Retry once only when the error says recovery is possible. Keep the original error in the audit record.

End every session with a record

```
onSessionEnd: async (input, invocation) => {
  audit.endReason = input.reason;
  audit.endedAt = new Date().toISOString();
  sessionState.delete(invocation.sessionId);

  return {
    sessionSummary: `Readiness session ended: ${input.reason}`,
  };
},
```

Cleanup should be safe to run more than once. A process crash may skip the hook, so the outer `finally` block still matters.

Test the control path, not model phrasing

Test	Stable assertion
`CR-1042`	Tool returns `ready` with no gaps
`CR-2087`	Tool returns `needs-review` with fixed gaps
Restricted ID	Pre-tool hook returns `deny` with the required reason
Failed tool event	Failed-tool hook records the failure
SDK error	Error hook chooses the expected recovery action
Session completion	End hook clears state and records the reason
Streaming	Deltas render once and final output is not duplicated

Mock the SDK boundary. Keep synthetic data fixed.

Trace the same run outside the terminal

```
CLI process
   | OTLP/HTTP :4318
   ▼
OpenTelemetry Collector
   | OTLP
   ▼
Jaeger :16686
```

```ts
const client = new CopilotClient({
  telemetry: {
    otlpEndpoint: "http://localhost:4318",
  },
});
```

The trace should show session work, tool activity, the denied action, and the end reason for one run.

Inspect the run in Jaeger

Use Jaeger to check:

- which span contained the tool request;
- whether the denied action reached the handler;
- how long the model and tool steps took;
- where a failed run stopped;
- whether one CLI invocation produced one connected trace.

Trace data can contain prompts and tool details. Keep this lab synthetic.

Reliability has tradeoffs

Choice	Benefit	Cost
Stream deltas	Faster visible feedback	More rendering state
Deny by default	Smaller execution boundary	More permission code
Hooks for audit	One lifecycle view	Inline latency if hooks do too much
Deterministic tool data	Stable tests and demos	Less realism
Local tracing stack	Traces you can inspect	Docker setup and local resource use

Choose controls that show what failed and why. Checks that only make the code look safer do not help.

Prepared demonstration

1. Run the passing tests.
2. Start the Collector and Jaeger.
3. Assess a known synthetic change.
4. Watch the response stream.
5. Request the restricted synthetic record.
6. Show the denial in the terminal.
7. Open the same run in Jaeger.
8. Trigger an unknown ID and show the failure path.

Keep the boundary tight

This session does not use:

- dynamic workflows or fleet;
- cloud or remote sessions;
- BYOK;
- plugin packaging;
- production deployment;
- external business systems;
- a second implementation language.

The lab is about one local TypeScript application's control path.

Lab handoff

Build the CLI in `lab/starter/sdk-readiness-assistant/`.

You will complete the lifecycle, streaming, readiness tool, permissions, hooks, tests, and OTLP export. Then you will run the known, denied, and failed paths and inspect the trace in Jaeger.

Stop if authenticated Copilot access, Node.js/npm, or Docker is unavailable.