ACCELQ

SAP TESTING GUIDE

Integrating test automation with SAP Cloud ALM: why it matters and how to build it

Why connecting your test automation tool directly to SAP Cloud ALM strengthens the whole toolchain, how Cloud ALM's Test Automation API works, and a hands-on build of an adapter from the documented APIs, with ACCELQ as the worked example.

Practitioner Reviewed by: Matt Angerer

October 18, 2026
www.accelq.com

INSIDE THIS GUIDE

Contents

01 Why integration matters now 4

02 How Cloud ALM's test automation model works 5

03 Benefits for the toolchain, by stakeholder 8

04 The ACCELQ example, day to day 10

05 Native integration vs building your own 13

06 How-to: build the integration 14

07 Build it with Claude Code: copy-paste prompt 24

08 Pitfalls 27

09 A 30-day rollout plan 28

10 Frequently asked questions 30

11 Summary and next steps 31

12 Sources 32

13 ACCELQ for SAP testing 35

SAP TESTING GUIDE

Integrating test automation with SAP Cloud ALM: why it matters and how to build it

Practitioner Reviewed by: Matt Angerer · October 18, 2026 · 32 min read

Most SAP programs that move to Cloud ALM end up with two systems that each think they own testing. Cloud ALM holds the processes, requirements, test plans and defects. The automation tool holds the scripts, the run history and the logs. If nothing connects them, someone copies results from one to the other by hand, usually late on a Thursday before a steering committee. This guide explains why a direct integration is worth the effort, how SAP's Test Automation API works, and how to build an integration yourself from both tools' documented APIs. ACCELQ is the worked example; the design applies to any provider.

KEY TAKEAWAYS

- Cloud ALM doesn't execute automated tests. It orchestrates and reports; an external provider implements SAP's Test Automation API and Cloud ALM calls it to sync test cases, trigger runs and fetch status.
- The direction matters: for the Test Automation API, your tool (or an adapter in front of it) is the server and Cloud ALM is the client. Cloud ALM's other public APIs work the other way around.
- The payoff is one system of record: process to requirement to test to result to defect, with automated results feeding the same traceability, analytics and quality gates as manual ones.
- If your tool has no native Cloud ALM connector, a small adapter on SAP BTP can implement the provider API and call the tool's REST API. ACCELQ's documented run, status and scenario APIs are enough to build one.
- Decide early which tool owns the test case master, how IDs map, and who creates defects. Most integration pain comes from those three decisions, not from code.
- Treat the adapter as production software: secrets in the BTP credential store, idempotent execution, retries, logs, and tests against non-production tenants.

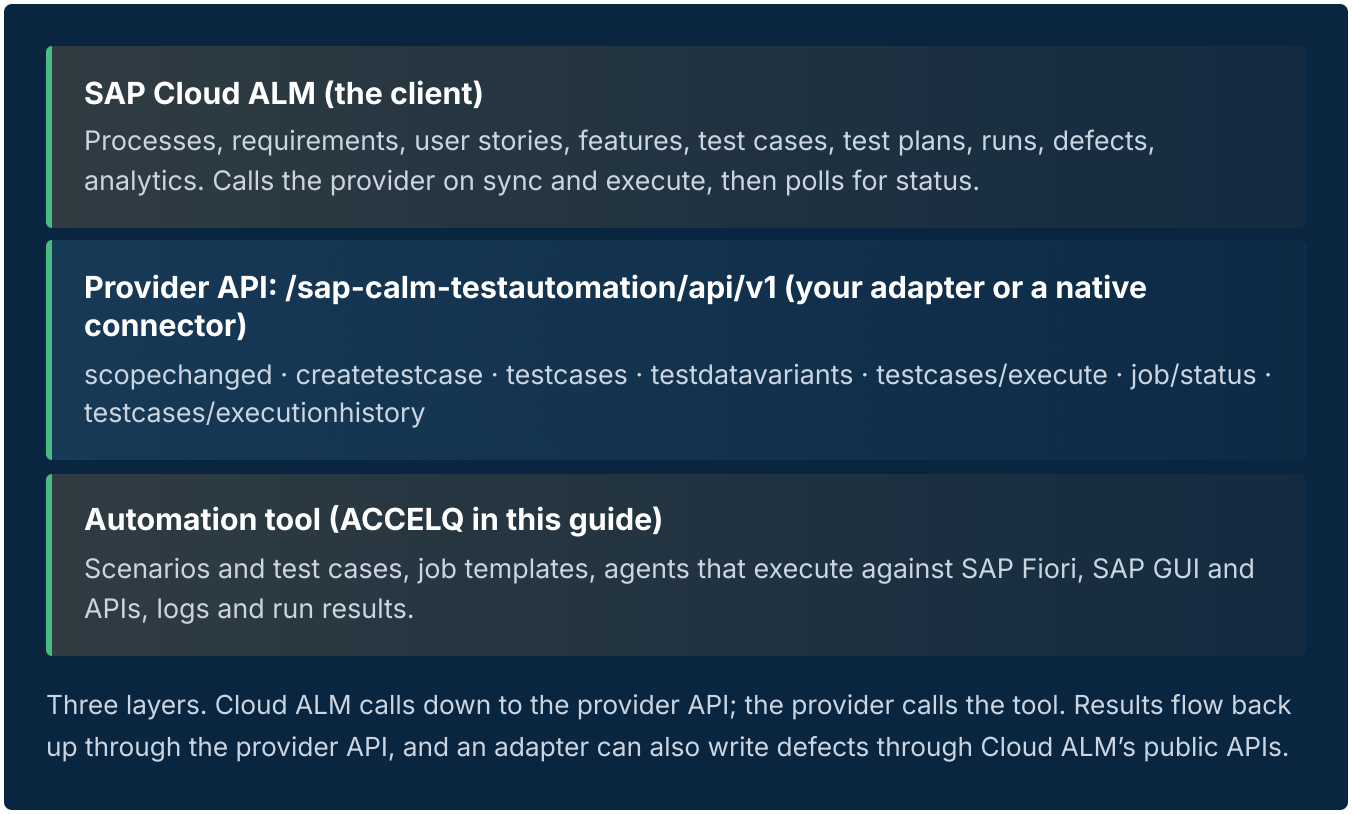
Why integration matters now

Three things changed at once for SAP testing teams. First, the calendar: SAP recommends completing the move from Solution Manager to Cloud ALM before Solution Manager 7.2 leaves mainstream maintenance at the end of 2027 [3]. Second, the automation engine: SAP says Component-based Test Automation (CBTA) is supported as long as Solution Manager is, but will not be expanded to SAP's cloud products, and that Cloud ALM does not provide an SAP home-grown automation engine [1]. SAP's position is that its ALM platforms stay open for third-party test automation integration [1]. Third, the test assets: SAP's transition guidance says that for S/4HANA private cloud, on-premise and Fiori apps, automation tools integrated with Cloud ALM need new test cases because of major UI changes [2].

Put together, that means the integration between Cloud ALM and an external automation tool is not a nice-to-have. It is the go-forward path for automated testing in SAP's ALM. If you are still setting up Cloud ALM itself, start with our practical guide to getting started with SAP Cloud ALM; its test automation section covers the options SAP lists. This guide picks up where that one stops.

How Cloud ALM's test automation model works

Cloud ALM integrates SAP and third-party automation providers that implement SAP's public Test Automation API, and adds orchestration, execution monitoring and reporting on top. The provider is responsible for implementing the API correctly and supporting it [5]. SAP's API catalog describes it as the interface an external tool must implement: Cloud ALM calls it to synchronize test cases, get data variants, trigger execution, check job status and retrieve history [10].



Three layers. Cloud ALM calls down to the provider API; the provider calls the tool. Results flow back up through the provider API, and an adapter can also write defects through Cloud ALM's public APIs.

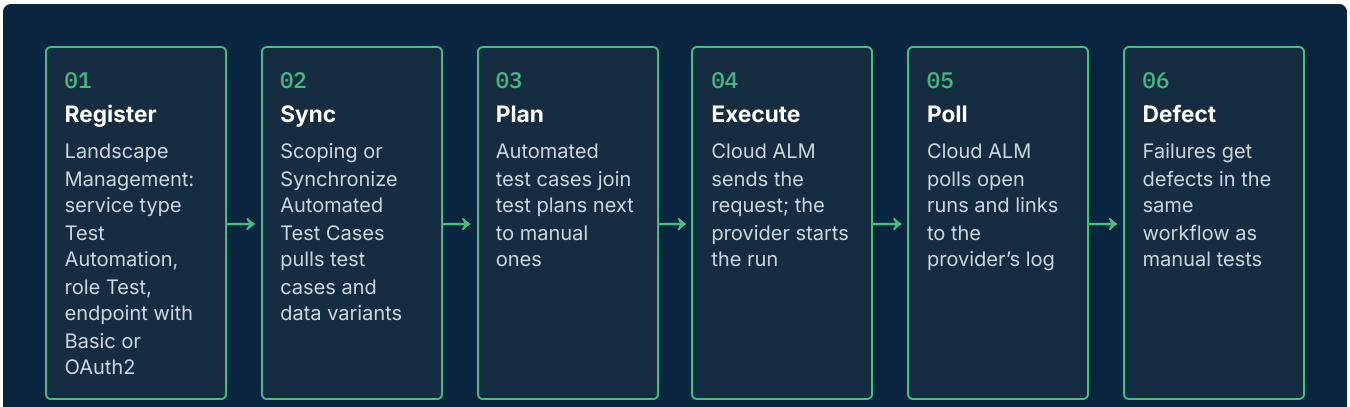
The provider endpoints

The specification is published as CALM_TEST_AUTOMATION on the SAP Business Accelerator Hub; version 1.0.13 was current when this guide was written [11]. It defines the provider-hosted base path **`/sap-calm-testautomation/api/v1`** and these operations [11]:

Operation	What Cloud ALM uses it for
`POST /scopechanged`	Tells the provider that process scope changed, which is the trigger for a test case sync
`/createtestcase`	Asks the provider to create a test case (check the method and body in the spec)
`GET /testcases`	Retrieves the provider's automated test cases, with a **`customFlag`** parameter
`GET /testdatavariants`	Retrieves data variants; each one becomes its own test case in Cloud ALM

Operation	What Cloud ALM uses it for
`POST /testcases/execute`	Starts execution of test cases from a test plan
`GET /job/status`	Polls the status of open runs
`GET /testcases/executionhistory`	Retrieves run history and results

The lifecycle, step by step



How an automated test moves through Cloud ALM once a provider is registered.

1. **Register.**A Global Administrator adds a service of type Test Automation in Landscape Management with role Test and the provider's root URL, then creates an endpoint with use case Test Automation, authenticated by basic authentication or OAuth2 client credentials [6].
2. **Synchronize.**Scoping processes triggers a sync of the provider's test cases [6]. Testers can also choose Synchronize Automated Test Cases in Test Preparation for a scope and provider, and each data variant becomes its own test case [7].
3. **Plan.** Automated test cases go into the same test plans as manual ones [8], so a SIT or regression cycle can mix both and report on them together [27].
4. **Execute and poll.**Execute sends the request to the provider; Cloud ALM polls open runs, updates status and results, and links to the provider's execution log [8].
5. **Defects.** Failed runs get defects from the test case list like any other failure, so automated and manual failures share one defect workflow [24],[25].

Where things show up in Cloud ALM

Artifact	Where you see it
Synced automated test cases and data variants	Test Preparation, under the scope and provider you synchronized [7],[9]

Artifact	Where you see it
Planned automated tests	Test Plans, alongside manual test cases [27]
Run status, results and the provider log link	Test Execution [8]
Defects from failed runs	Defects, linked from the run [25],[26]
Coverage and status by requirement or process	Requirement Traceability and Solution Process Traceability [29],[30]
Cycle progress and trends	Test Execution Analysis [31]

The other direction: Cloud ALM's public APIs

Separately from the provider API, Cloud ALM publishes APIs that your code calls, all on the Business Accelerator Hub [12]. The ones that matter for a test automation integration:

- **Test Management** (`/api/calm-testmanagement/v1`): manual test cases with activities, actions and references, created in one call to `ManualTestCases`; updates use `PATCH` with an `If-Match` header [13],[18].
- **Test Plans** (`/api/calm-testmanagement-testplans/v1`), marked beta on the hub [14].
- **Tasks** (`/api/calm-tasks/v1/tasks`): tasks, user stories, requirements and defects are all task types (`CALMTASK`, `CALMUS`, `CALMREQU`, `CALMDEF`) [15],[19],[20].
- **Projects** (`/api/calm-projects/v1/projects`), **features** (`/api/calm-features/v1`) and process scopes (`/api/calm-processmanagement/v1`) [17],[18],[20].
- **Analytics** (`/api/calm-analytics/v1`), built on providers, dimensions and metrics, which SAP's own Grafana plug-in uses [16],[23].

Authentication for these APIs is OAuth 2.0 client credentials from an SAP Cloud ALM API service instance in your BTP subaccount. You create the instance with the scopes you need, then create a service key; the key inherits the instance's scopes [21],[22].

Benefits for the toolchain, by stakeholder

The core argument is simple: an automated result that lives only in the automation tool is invisible to the people who make release decisions in Cloud ALM. Integration makes it visible, in context, without anyone retyping it.

Test manager

- **One test plan for manual and automated work.** Cycle progress, testers and automated runs sit in one plan, so the status you report is the status the tool shows [27],[8].
- **No swivel-chair status copying.** Cloud ALM polls the provider and updates results itself [8]. The hours spent transcribing pass/fail counts go back into test design.
- **One defect workflow.** Automated failures enter the same triage, severity and retest loop as manual ones [25],[26]. Pair it with our defect severity triage matrix.

Release manager

- **Feature and transport readiness.**Features carry transports through the landscape; when automated regression results sit next to them in the same tool, the “can this feature move?” question has evidence behind it [33].
- **Regression on the vendor’s schedule.** Public Edition upgrades and Private Edition feature and support pack stacks arrive on a fixed cadence. Triggering a regression plan from Cloud ALM and reading results there turns each upgrade into a repeatable cycle. Our upgrade regression guide covers the cadence.
- **Quality gates with data.** Cloud ALM quality gates are where go/no-go decisions are recorded [32]. Automated coverage and results feed them directly; our go/no-go readiness checklist lists what to check.

Auditor and compliance

- **End-to-end traceability in one system of record.** Requirement and solution process traceability show which tests cover which requirement or process, with their latest results [29],[30]. With integration, that includes automated evidence, linked to the provider’s log.
- **SOX and change evidence.**For in-scope changes, auditors usually want to see that the change was tested, by whom or by what, with what result, and that failures were resolved before release. An integrated toolchain answers that from one place instead of screenshots stitched together from two. Confirm with your auditors which artifacts they accept and how long you must retain the provider’s logs, because Cloud ALM links to them rather than storing them.

- **Governance of AI-generated tests.** Joule can generate manual test cases in Cloud ALM [34], and many automation tools now generate tests too. Routing generated tests through Cloud ALM preparation statuses, test plans and traceability gives you one review point for human sign-off, whoever or whatever wrote the test [28].

CIO and platform owner

- **Analytics without a side database.**Automated results land in Cloud ALM's analytics, which the Analytics API exposes to dashboards outside Cloud ALM too [31],[16],[23].
- **A standard interface, not a one-off.**The provider API is public and versioned, so the integration follows SAP's published contract rather than screen scraping or file drops [11],[10].
- **The cost of not integrating.** Without integration you pay in hours of status copying, in late-discovered failures that never reached the defect list, in go/no-go meetings that debate whose numbers are right, and in audit requests answered by hand. Those costs are real even if nobody budgets them; estimate them for your own program before you decide.

The ACCELQ example, day to day

ACCELQ describes its platform as codeless automation across SAP products and their integrations [45]. Its public API reference documents what an integration needs [36],[39],[40]:

- **Authentication:** every call sends a **`user_id`** header (the login user) and an **`api_key`**header (the user's API key). Tenant and project codes are shown under Auth properties in the user profile [36],[39].
- **Run now:** **`POST {url}/awb/api/2.0/{tenantCode}/{projectCode}/test-exec/jobs/run`** triggers a scenario or test suite; the job is queued until a suitable agent is free, and the response returns a **`jobPid`** [36],[41].
- **Run a template:** **`POST …/test-exec/jobs/templates/{templateJobId}/run`** runs a saved job configuration with optional run-property overrides, and **`GET …/test-exec/jobs/templates`** lists templates [37],[38].
- **Status and results:** **`GET {url}/awb/api/1.0/{tenantCode}/runs/{runPid}`** returns the job status, pass, fail and not-run counts, and a per-test-case list with status, error message, timestamps and a result link [39].
- **Scenarios:** **`POST …/scn/scenarios/{auto|manual}/fetch-details`** returns scenarios with tags, custom fields and, optionally, requirement traceability, up to 25 per page [40].
- **CI/CD and agents:**ACCELQ's GitHub Actions and Jenkins plugins trigger saved CI jobs and poll the same run-status endpoint every 30 seconds [42],[43]. Jobs run on ACCELQ agents, including Dockerized agents you host inside your network [44], and the run API can target agents by name or availability [36]. Run notifications go to email, Slack or Teams channels [36].

With an adapter in place, a regression cycle looks like this:

1. The test lead scopes the order-to-cash processes in Cloud ALM; the sync pulls in the mapped ACCELQ scenarios as automated test cases.
2. She adds them to the "2608 upgrade regression" test plan next to the manual UAT cases and assigns owners.
3. She chooses Execute. The adapter starts the ACCELQ job template for that plan on the agent pool inside the network.
4. Cloud ALM polls; the adapter reads the run status and reports each test case as passed or failed with a link to the ACCELQ result.
5. Two cases fail. Defects appear in Cloud ALM with the error message and log link, and go into the normal triage meeting.

6. The release manager opens the feature, sees the failed regression and open defects, and holds it until retest passes.

Nobody exported a spreadsheet, and the auditor's question three months later is answered from the traceability view.

What ACCELQ brings to a Cloud ALM program today

The integration moves results; the automation behind it decides whether there are results worth moving. Here is what each common pain costs a Cloud ALM program, and what ACCELQ documents today to address it.

Pain point	What it costs	How ACCELQ addresses it today
Traceability for sign-off	Quality gates and auditors ask which requirements have an executed, passing test. If automated evidence lives only in the tool, the answer is assembled by hand before every gate.	Scenarios link to requirements in Jira, Azure DevOps or Rally, with requirement-based suites [46],[47]; through an adapter like the one below, the same results reach Cloud ALM's traceability views. Native SAP Cloud ALM integration is in development (roadmap per ACCELQ; not generally available).
Regression on SAP's release cadence	Every upgrade or support pack stack needs the same regression again. Run by hand, the cycle competes with project work and coverage shrinks.	Data-driven test cases and dynamic suites by tag, module or requirement [52],[55], started by job templates from CI or the API [37],[36].
Rebuilding tests after UI changes	SAP expects new test cases where UIs changed [2], and every later Fiori or support pack change moves screens again. Brittle scripts eat the automation team's time.	Self-healing recovers broken element identification at runtime and reports what it changed [54]. It repairs how elements are found, not changed process logic.
Business users running manual tests	UAT and FUT testers outside Cloud ALM produce evidence in spreadsheets and email that someone has to transcribe.	The manual Test Runner records step status, screenshots and Jira or Azure DevOps (TFS) defects [48], and a manual scenario links to its automated counterpart [49]. Cloud ALM's own manual test management remains an option for testers who work there.
Cross-platform end-to-end tests	An SAP process that crosses web apps, APIs and queues gets tested in pieces, so the integration defects surface late.	One scenario across SAP GUI, web, mobile, API and database steps [50],[51]. The SAP GUI framework is documented for ECC 6.0 and higher [56]; S/4HANA and Fiori coverage is claimed in SAP's partner listing (vendor-supplied text) [57], so pilot your own apps.
Turning requirements into tests	Fit-to-standard produces more requirements than a test team can design by hand before the first cycle.	Autopilot generates test cases from scenario logic [52], and the MCP server lets an AI assistant propose a test plan for approval before it creates tests [53]. Availability depends on plan. Generating tests from Cloud ALM requirements is part of the in-development native integration.

TRY THE WORKED EXAMPLE

Build the adapter against a real ACCELQ tenant

Everything in the how-to below needs an ACCELQ user, an API key and a project with at least one scenario. The free trial gives you all three, so you can prove the run trigger and status polling before you touch Cloud ALM.

Start the free ACCELQ trial

Native integration vs building your own

SAP's documentation lists the SAP test automation tool for S/4HANA Cloud Public Edition, a pre-integrated option for browser-based SAP applications, and other providers that implement the API [5]. Some providers ship a ready-made connector. ACCELQ's native SAP Cloud ALM integration is in development (roadmap per ACCELQ; not generally available). As described by ACCELQ, it will consume requirements and fit/gap decisions from Cloud ALM and generate the matching tests; as of October 24, 2026 it is not in ACCELQ's public documentation or on SAP's partner page [47], [57]. Until it ships, an adapter built from the documented APIs is the way to connect the two. Ask ACCELQ for the current status before you build, because a supported connector beats one you maintain.

Question	Use a native connector when...	Build an adapter when...
Availability	Your tool vendor ships and supports one for the provider API	No connector exists, or it lacks a capability you need
Support	You want the vendor to own API changes (SAP makes the provider responsible for the implementation)	You can own a small service and its on-call
Mapping	The vendor's default mapping of tests and variants fits your structure	You need custom rules: job templates per plan, tags, data variants, multiple projects
Defects	You are happy with defects created from the Cloud ALM run list	You want policy-driven defect creation with your own fields and deduplication
Network	The vendor's cloud can reach your execution agents	You need to keep the bridge inside your own BTP subaccount

Whichever you choose, run the same proof points from our Cloud ALM guide's automation checklist in your own tenant before you commit, and score options with the tool evaluation scorecard.

How-to: build the integration

> ⚠ **Everything in this section is illustrative**
>
> The code samples show the shape of a working adapter using only endpoints I could confirm in SAP's and ACCELQ's published references in October 2026. Where a schema or field could not be confirmed, the code uses a clearly named placeholder and a TODO pointing to the reference. Check the current specs before you build, and never point a first version at a production tenant.

Prerequisites

BEFORE YOU START

- A Cloud ALM tenant with a Global Administrator who can edit Landscape Management
- A BTP subaccount where you can create an SAP Cloud ALM API service instance, plus a Cloud Foundry space or Kyma namespace for the adapter
- An ACCELQ project with at least one automated scenario, a user with an API key, and an agent that can reach your SAP test system
- A non-production Cloud ALM project and SAP test client to integrate against first
- Written decisions on test case ownership, ID mapping and defect policy (see below)
- Access to the CALM_TEST_AUTOMATION and CALM_TKM specs on the Business Accelerator Hub

Decide three things before you write code

1. **Who owns the test case master.** For automated tests, the script lives in the automation tool and Cloud ALM holds a synced reference. Make that explicit: test design changes happen in ACCELQ, and Cloud ALM is where tests are planned, executed and traced. Two-way editing of the same test case is how sync conflicts start.
2. **How IDs map.**Decide whether one Cloud ALM automated test case equals one ACCELQ scenario, one scenario test case, or one job template. Keep the map in a versioned config file, not in people's heads.

3. **Who creates defects.** Either testers create them from the Cloud ALM run list after triage (simplest, and the documented flow [25]), or the adapter creates them automatically under a policy you agree with the test manager. Automatic defects without deduplication flood the backlog.

Architecture choice

Run a small adapter service in your own BTP subaccount. It implements the provider API that Cloud ALM calls, calls ACCELQ's REST API, and optionally calls Cloud ALM's public APIs for defects. Node.js on Cloud Foundry is the lowest-friction option; Kyma suits teams already running Kubernetes workloads on BTP. Either way the adapter needs a small store (for example PostgreSQL on BTP, or Redis) for idempotency keys and the mapping between Cloud ALM execution requests and ACCELQ job IDs.

Step 1: create the Cloud ALM API service instance and key

1. In the BTP subaccount linked to Cloud ALM, create a service instance of SAP Cloud ALM API (plan standard).
2. Pass the scopes in the instance parameters, not the key; the key inherits them [21]. Ask only for what the adapter needs, for example task write access if it creates defects and test management read access if it reads test cases. Take the exact scope names from the API reference for your tenant [12],[22].
3. Create a service key of type client ID and secret. It contains the client ID, client secret and the token URL base.
4. Store the key in the BTP credential store or bind it to the adapter; do not download it to laptops or commit it.

Step 2: create ACCELQ API credentials

Use a dedicated integration user in ACCELQ rather than a person's login, with access only to the projects the adapter runs. Copy the API key, tenant code and project code from the user profile's Auth properties [36], and store the key in the same secret store. Rotate it on a schedule and when people with access leave.

Step 3: build the adapter

Configuration comes from the environment. Nothing secret lives in the repository.

CONFIG.TS (ILLUSTRATIVE)

```
// config.ts — ILLUSTRATIVE. Values come from the environment or a bound service,
// never from source control. Field names follow a standard BTP service key; check yours.
export type CalmServiceKey = {
  clientid: string
  clientsecret: string
  url: string // XSUAA authentication URL from the SAP Cloud ALM API service key
}

export const cfg = {
  // e.g. https://<tenant>.<region>.alm.cloud.sap/api
  calmApiBase: process.env.CALM_API_BASE as string,
  calmKey: JSON.parse(process.env.CALM_SERVICE_KEY ?? '{}') as CalmServiceKey,
  // ACCELQ login URL, tenant and project codes come from the user profile (Auth properties)
  aqUrl: process.env.ACCELQ_URL as string,
  aqTenant: process.env.ACCELQ_TENANT_CODE as string,
  aqProject: process.env.ACCELQ_PROJECT_CODE as string,
  aqUserId: process.env.ACCELQ_USER_ID as string, // login user (email)
  aqApiKey: process.env.ACCELQ_API_KEY as string,
}
```

Token fetch. Client credentials against the authentication URL from the service key, with caching [21], [18]:

CALMTOKEN.TS (ILLUSTRATIVE)

```
// calmToken.ts — ILLUSTRATIVE. OAuth 2.0 client credentials against the XSUAA
// instance behind your SAP Cloud ALM API service key. Cache the token; don't fetch per call.
import { cfg } from './config'

let cached: { token: string; expiresAt: number } | null = null

export async function getCalmToken(): Promise<string> {
  if (cached && Date.now() < cached.expiresAt - 60_000) return cached.token
  const { clientid, clientsecret, url } = cfg.calmKey
  const res = await fetch(url + '/oauth/token', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      Authorization: 'Basic ' + Buffer.from(clientid + ':' + clientsecret).toString('base64'),
    },
    body: new URLSearchParams({ grant_type: 'client_credentials' }),
  })
  if (!res.ok) throw new Error('Cloud ALM token request failed: HTTP ' + res.status)
  const body = (await res.json()) as { access_token: string; expires_in: number }
  cached = { token: body.access_token, expiresAt: Date.now() + body.expires_in * 1000 }
  return cached.token
}

export async function calm(path: string, init: RequestInit = {}): Promise<Response> {
  const token = await getCalmToken()
  return fetch(cfg.calmApiBase + path, {
    ...init,
```

```
    headers: { Authorization: 'Bearer ' + token, Accept: 'application/json', 'Content-Type':
'application/json', ...init.headers },
  })
}
```

Run trigger. Use the run API for a single scenario, or the template API when a test plan maps to a saved job configuration. Templates are easier to govern, because agent selection, environment and notifications are set once in ACCELQ [36],[37].

ACCELQ.TS (ILLUSTRATIVE)

```
// accelq.ts — ILLUSTRATIVE. Endpoints and headers from ACCELQ's external API reference.
import { cfg } from './config'

function aqHeaders(): Record<string, string> {
  return {
    user_id: cfg.aqUserId,
    api_key: cfg.aqApiKey,
    'Content-Type': 'application/json',
    Accept: 'application/json',
  }
}

const v2 = () => cfg.aqUrl + '/awb/api/2.0/' + cfg.aqTenant + '/' + cfg.aqProject

/** Run one scenario now. POST {url}/awb/api/2.0/{tenantCode}/{projectCode}/test-exec/jobs/run */
export async function runScenario(scnPid: string, purpose: string, runProperties: Record<string, string> =
{}) {
  const res = await fetch(v2() + '/test-exec/jobs/run', {
    method: 'POST',
    headers: aqHeaders(),
    body: JSON.stringify({ runEntityType: 'SCENARIO', scnPid, purpose, runProperties }),
  })
  if (!res.ok) throw new Error('ACCELQ run failed: HTTP ' + res.status + ' ' + (await res.text()))
  return (await res.json()) as { jobPid: number; message: string }
}

/** Run a saved job template. POST .../test-exec/jobs/templates/{templateJobId}/run */
export async function runTemplate(templateJobId: number, runProperties: Record<string, string> = {}) {
  const res = await fetch(v2() + '/test-exec/jobs/templates/' + templateJobId + '/run', {
    method: 'POST',
    headers: aqHeaders(),
    body: JSON.stringify({ runProperties }),
  })
  if (!res.ok) throw new Error('ACCELQ template run failed: HTTP ' + res.status + ' ' + (await
res.text()))
  return (await res.json()) as { jobPid: number; templateJobPid: number; message: string }
}
```

Status polling. The run-status response carries the job status and a per-test-case list [39]. ACCELQ's CI plugins poll every 30 seconds and treat completed, aborted, failed to start and error as finished states [42],[43]. Confirm in your tenant that the **`jobPid`** returned by the run call is the **`runPid`** accepted by the status call; the reference uses both names.

POLL.TS (ILLUSTRATIVE)

```
// poll.ts — ILLUSTRATIVE. GET {url}/awb/api/1.0/{tenantCode}/runs/{runPid}
import { cfg } from './config'

export type AqTestCase = {
  scnPid: number
  dbPid: number // test case id
  name: string
  status: string // documented values include pass, fail, notRun, running, info, fatal, warn
  lastErrorStmtMsg: string
  startDate: number
  endDate: number
  extResultLink: string
}
export type AqSummary = {
  jobPid: number
  status: string
  pass: number
  fail: number
  notRun: number
  testcaseCount: number
  testCaseSummaryList: AqTestCase[]
}

// The CI plugins treat these job states as finished. Compare case-insensitively:
// the API reference example returns lowercase "completed".
const DONE = ['completed', 'aborted', 'failed to start', 'error']

export async function getRun(runPid: number): Promise<AqSummary> {
  const res = await fetch(cfg.aqUrl + '/awb/api/1.0/' + cfg.aqTenant + '/runs/' + runPid, {
    headers: { user_id: cfg.aqUserId, api_key: cfg.aqApiKey, Accept: 'application/json' },
  })
  if (!res.ok) throw new Error('ACCELQ status failed: HTTP ' + res.status)
  return ((await res.json()) as { summary: AqSummary }).summary
}

export async function waitForRun(runPid: number, maxMinutes = 120): Promise<AqSummary> {
  const deadline = Date.now() + maxMinutes * 60_000
  let delay = 30_000 // ACCELQ's own CI plugins poll every 30 seconds
  for (;;) {
    const s = await getRun(runPid)
    if (DONE.includes(s.status.toLowerCase())) return s
    if (Date.now() > deadline) throw new Error('Run ' + runPid + ' still ' + s.status + ' after ' +
maxMinutes + ' min')
    await new Promise((r) => setTimeout(r, delay))
    delay = Math.min(delay * 1.5, 120_000)
  }
}
```

Result mapping. ACCELQ test case statuses (pass, fail, notRun, running, info, fatal, warn) [42],[39] must map to the status values the provider API returns to Cloud ALM. Take those values from the spec; the sample uses placeholders on purpose.

MAPRESULT.TS (ILLUSTRATIVE)

```
// mapResult.ts — ILLUSTRATIVE. Map ACCELQ test case results to the status values
// your Cloud ALM provider API responses must return.
import type { AqSummary } from './poll'

// TODO: replace with the exact status enumeration in the CALM_TEST_AUTOMATION spec
// (Business Accelerator Hub, job status and execution history responses).
type CalmStatus = 'CALM_STATUS_PASSED' | 'CALM_STATUS_FAILED' | 'CALM_STATUS_NOT_RUN' |
'CALM_STATUS_RUNNING'

const STATUS: Record<string, CalmStatus> = {
  pass: 'CALM_STATUS_PASSED',
  info: 'CALM_STATUS_PASSED',
  warn: 'CALM_STATUS_PASSED', // a policy choice: decide with your test manager
  fail: 'CALM_STATUS_FAILED',
  fatal: 'CALM_STATUS_FAILED',
  notrun: 'CALM_STATUS_NOT_RUN',
  running: 'CALM_STATUS_RUNNING',
}

export function mapRun(summary: AqSummary) {
  return summary.testCaseSummaryList.map((tc) => ({
    // Your mapping key: ACCELQ scenario id + test case id -> Cloud ALM automated test case
    externalId: tc.scnPid + ':' + tc.dbPid,
    name: tc.name,
    status: STATUS[tc.status.toLowerCase()] ?? 'CALM_STATUS_FAILED', // unknown = fail loudly
    startedAt: new Date(tc.startDate).toISOString(),
    endedAt: tc.endDate ? new Date(tc.endDate).toISOString() : null,
    message: tc.lastErrorStmtMsg || undefined,
    logUrl: tc.extResultLink, // Cloud ALM links out to the provider's log
  }))
}
```

The provider endpoints. This is the part Cloud ALM calls. The paths come from the spec [11]; the bodies must too.

SERVER.TS (ILLUSTRATIVE SKELETON)

```
// server.ts — ILLUSTRATIVE skeleton of the provider side: the endpoints Cloud ALM calls.
// Paths are from the CALM_TEST_AUTOMATION spec (v1.0.13). Request and response bodies are
// deliberately left as TODOs: copy them from the spec, don't infer them from this sample.
import express from 'express'

const app = express()
app.use(express.json())
const base = '/sap-calm-testautomation/api/v1'
```

```
// TODO: authenticate Cloud ALM's calls (Basic or OAuth2 client credentials, matching
// the endpoint you register in Landscape Management) before any route below.

app.post(base + '/scopechanged', async (req, res) => {
  // Cloud ALM notifies the provider that process scope changed. TODO: schema per spec.
  res.status(200).json({})
})
// The spec also defines /createtestcase; check its method and body in the spec.

app.get(base + '/testcases', async (req, res) => {
  // Return the automated test cases Cloud ALM may sync (spec defines a customFlag parameter).
  // Source: ACCELQ Scenario API, POST .../scn/scenarios/auto/fetch-details (max 25 per page).
  res.json(/* TODO: map ACCELQ scenarios to the spec's test case schema */ [])
})

app.get(base + '/testdatavariants', async (req, res) => {
  // Each data variant becomes its own test case in Cloud ALM. TODO: schema per spec.
  res.json([])
})

app.post(base + '/testcases/execute', async (req, res) => {
  // 1. Look up the ACCELQ scenario or job template for each requested test case (mapping file)
  // 2. runScenario() or runTemplate(); persist jobPid against Cloud ALM's execution id (idempotency)
  // 3. Return the job reference in the shape the spec defines
  res.status(202).json({})
})

app.get(base + '/job/status', async (req, res) => {
  // Cloud ALM polls open runs. getRun(jobPid) -> mapRun() -> spec status shape.
  res.json({})
})

app.get(base + '/testcases/executionhistory', async (req, res) => {
  res.json([])
})

app.listen(Number(process.env.PORT ?? 8080))
```

Step 4: register the endpoint in Cloud ALM

1. Deploy the adapter to a non-production space and note its public root URL.
2. In Landscape Management, add a service of type Test Automation with role Test and the adapter's root URL. Add an endpoint with use case Test Automation, using basic authentication or OAuth2 client credentials that the adapter validates [6].
3. Check the endpoint in Landscape Management, then watch the adapter logs for the first inbound call.

Step 5: map and synchronize test cases

Populate the mapping file from ACCELQ's scenario API: filter by tag (for example a **`calm-sync`** tag) or by requirement keys, page through 25 at a time, and return only mapped scenarios from **`/testcases`** [40]. In Cloud ALM, run Synchronize Automated Test Cases for one scope and provider and confirm the names and data variants read well to a functional tester [7].

Step 6: trigger a run from a test plan

Add a synced test case to a test plan and choose Execute [8]. In the adapter, the execute handler looks up the mapping, starts the ACCELQ run or template, and stores the job ID against Cloud ALM's request so a retry does not start a second run.

Step 7: post results back

Results flow back through the provider API: Cloud ALM polls **`/job/status`** and reads **`/testcases/executionhistory`**, and the adapter answers from ACCELQ's run status, mapped to Cloud ALM's statuses, with ACCELQ's result link as the log link [11],[8],[39].

Step 8: defect creation

If you automate defects, create them through the Tasks API as type **`CALMDEF`** [15],[20], once per failed test case per run, with the error message, the ACCELQ log link and an adapter key for deduplication.

DEFECT.TS (ILLUSTRATIVE)

```
// defect.ts — ILLUSTRATIVE. Defects are a task type in Cloud ALM's Tasks API.
// POST {CALM_API_BASE}/calm-tasks/v1/tasks with type CALMDEF (type code per SAP's sample notebooks).
// TODO: confirm required fields (priority, status) and any test case or run link fields in the
// CALM_TKM spec before relying on this; Cloud ALM's UI flow "create defect from test run" may
// link records that this call does not.
import { calm } from './calmToken'

export async function createDefect(input: {
  projectId: string
  testCaseName: string
  message?: string
  logUrl: string
  idempotencyKey: string // e.g. runPid + ':' + test case id
}) {
  const res = await calm('/calm-tasks/v1/tasks', {
    method: 'POST',
    body: JSON.stringify({
      projectId: input.projectId,
      type: 'CALMDEF',
      title: '[Automated] ' + input.testCaseName + ' failed',
      description:
```

```
        (input.message ?? 'Automated test failed.') +
        '\nExecution log: ' + input.logUrl +
        '\nAdapter key: ' + input.idempotencyKey,
    }),
  })
  if (!res.ok) throw new Error('Cloud ALM defect create failed: HTTP ' + res.status + ' ' + (await
res.text()))
  return (await res.json()) as { id: string }
}
```

Security

- Keep the Cloud ALM service key and the ACCELQ API key in the BTP credential store or behind a BTP destination; read them at runtime.
- Never log tokens, keys or request bodies that may contain business data. Log IDs, statuses and timings.
- Authenticate every inbound call from Cloud ALM with the credentials you registered, and reject everything else.
- Use least-privilege scopes on the Cloud ALM API instance and a dedicated, project-limited ACCELQ user.
- Rotate both credentials on a schedule, and immediately when someone with access leaves.
- Keep run properties free of production passwords; use the tool's own encrypted properties or test accounts.

Testing the integration itself

INTEGRATION TEST CHECKLIST

- ☑ Unit tests with mocked HTTP for token fetch and caching, run trigger, polling to every terminal state, status mapping and defect creation
- ☑ Contract tests that load example payloads from both API references, so a schema change fails a build, not a regression cycle
- ☑ Sync one scope end to end in a non-production project; check names, data variants and counts against ACCELQ
- ☑ Execute one passing and one failing test from a test plan; confirm statuses, timestamps and log links in Test Execution
- ☑ Send the same execute request twice; confirm only one ACCELQ run starts

- Stop the agent mid-run; confirm the run ends as aborted or error in Cloud ALM, not stuck as running
- Expire and revoke credentials on each side; confirm clear errors and no secrets in logs
- Run a test plan with dozens of cases; watch for 429s, timeouts and polling load on both sides

Operations and monitoring

- Alert on failed token fetches, ACCELQ 401 and 403 responses, and runs that stay in progress past their expected duration.
- Expose a health endpoint and monitor it from BTP or Cloud ALM's own monitoring.
- Review the mapping file with every new scope or test plan; an unmapped test case is a silent gap in coverage.
- Watch Cloud ALM's fair-use limits for API transfer volume per tenant [35], and keep polling intervals sensible.
- Re-check the CALM_TEST_AUTOMATION and ACCELQ API versions each quarter, and run the contract tests when either changes.

Build it with Claude Code: copy-paste prompt

If you use Claude Code, this prompt gives it everything above in one go: the verified endpoints on both sides, the open questions to ask you first, and the guardrails. Replace the placeholders in angle brackets as you answer its questions. Do not paste secrets into the prompt or the chat; supply them through environment variables locally and the BTP credential store when deployed.

CLAUDE CODE PROMPT: CLOUD ALM ↔ ACCELQ ADAPTER

```
You are helping me build a small integration adapter between SAP Cloud ALM and ACCELQ.
Work in a new, empty repository. Do not deploy anything, and do not call any real tenant, until I confirm.

## Before you write any code
Ask me for, and wait for my answers to:
1. My SAP Cloud ALM API base URL, in the form <CALM_TENANT_URL>/api (for example https://<tenant>.
<region>.alm.cloud.sap/api), and the Cloud ALM project ID(s) in scope: <CALM_PROJECT_ID>.
2. My ACCELQ login URL <ACCELQ_URL>, tenant code <TENANT_CODE> and project code <PROJECT_CODE>.
3. My deployment target: SAP BTP Cloud Foundry (default) or Kyma, plus the BTP region and space.
4. My test case mapping rules: how a Cloud ALM automated test case maps to an ACCELQ scenario, test case
or job template (for example by ACCELQ scenario ID, by a naming convention, or by an explicit table), and
how data variants should be handled.
5. My defect policy: create a Cloud ALM defect on every failed test case, only on first failure, or never
automatically; and who the default assignee is.
Never ask me to paste secrets (client secrets, API keys, passwords, tokens) into this chat. Secrets are
supplied at runtime through environment variables locally and through the BTP credential store or a
destination on BTP.

## What to build
A TypeScript service on Node.js 22 with:

### A. Provider API that SAP Cloud ALM calls
SAP Cloud ALM's Test Automation API (CALM_TEST_AUTOMATION on the SAP Business Accelerator Hub,
https://api.sap.com/api/CALM_TEST_AUTOMATION/overview) is implemented by the test automation provider and
called by Cloud ALM. Expose these routes under /sap-calm-testautomation/api/v1:
- POST /scopechanged
- /createtestcase (take the HTTP method from the spec)
- GET /testcases (supports the spec's customFlag parameter)
- GET /testdatavariants
- POST /testcases/execute
- GET /job/status
- GET /testcases/executionhistory
Take every request and response schema, parameter name and status value from the official spec. If you
cannot open or confirm a schema, write a typed placeholder and a TODO comment with the link
https://api.sap.com/api/CALM_TEST_AUTOMATION/overview instead of guessing.
Protect these routes with the authentication I will register in Cloud ALM Landscape Management for the
Test Automation endpoint (Basic or OAuth2 client credentials; ask me which).

### B. ACCELQ client (verified endpoints; all use headers user_id and api_key plus Content-Type and Accept
application/json)
- Run a scenario or test suite now: POST <ACCELQ_URL>/awb/api/2.0/<TENANT_CODE>/<PROJECT_CODE>/test-
exec/jobs/run with a body containing runEntityType (SCENARIO or TEST_SUITE), scnPid or testSuitePid,
purpose and runProperties. Response contains jobPid.
- Run a saved job template: POST <ACCELQ_URL>/awb/api/2.0/<TENANT_CODE>/<PROJECT_CODE>/test-
exec/jobs/templates/{templateJobId}/run with optional runProperties. Response contains jobPid and
templateJobPid.
```

```
- List job templates: GET <ACCELQ_URL>/awb/api/2.0/<TENANT_CODE>/<PROJECT_CODE>/test-exec/jobs/templates?
page=&pageSize= (max pageSize 100).
- List scenarios for test case sync: POST
<ACCELQ_URL>/awb/api/2.0/<TENANT_CODE>/<PROJECT_CODE>/scn/scenarios/auto/fetch-details?
pageNumber=&pageSize= (max pageSize 25).
- Poll a run: GET <ACCELQ_URL>/awb/api/1.0/<TENANT_CODE>/runs/{runPid}. The response has a summary object
with status, pass, fail, notRun, testcaseCount and testCaseSummaryList (each with scnPid, dbPid, name,
status, lastErrorStmtMsg, startDate, endDate, extResultLink).
Reference: https://accelq-docs.github.io/external-api-doc/ . Confirm that the jobPid returned by the run
call is the runPid accepted by the status call, and add a test that documents it.
Treat job status values completed, aborted, failed to start and error as terminal (case-insensitive). Map
test case statuses pass, fail, notRun, running, info, fatal and warn to the Cloud ALM status values from
the spec, using a mapping table I approve.

### C. Cloud ALM public API client
- OAuth 2.0 client credentials token from the SAP Cloud ALM API service key (clientid, clientsecret, url):
POST {url}/oauth/token with grant_type=client_credentials. Cache the token until shortly before expiry.
- Defects: Cloud ALM defects are tasks of type CALMDEF in the Tasks API, POST <CALM_TENANT_URL>/api/calm-
tasks/v1/tasks (CALM_TKM on the Business Accelerator Hub). Confirm required fields and any fields that
link a defect to a test case or run in the spec; mark anything unconfirmed as a TODO with the link
https://api.sap.com/api/CALM_TKM/overview.
- Optional reads: Test Management (/api/calm-testmanagement/v1), Test Plans (/api/calm-testmanagement-
testplans/v1, beta) and Analytics (/api/calm-analytics/v1). Use only operations you can confirm in the
spec.
- Request only the scopes the adapter needs when I create the service instance, and list them in the
README.

### D. Cross-cutting requirements
- Configuration: a mapping file (config/mapping.json or YAML) that maps Cloud ALM automated test case IDs
to ACCELQ scenario IDs, test case IDs or job template IDs, validated at startup with a schema.
- Secrets: read from environment variables locally and from the BTP credential store or a BTP destination
when deployed. Never hard-code secrets, never write them to logs, and add .env to .gitignore with a
committed .env.example that has placeholder values only.
- Retries with exponential backoff and jitter for 429 and 5xx responses on both APIs, with a cap; no retry
on 4xx other than 429.
- Idempotency: persist the link between Cloud ALM's execution request and the ACCELQ jobPid so a repeated
execute call does not start a second run, and a defect is created at most once per failed test case per
run.
- Structured JSON logging with a correlation ID per Cloud ALM request; log IDs and statuses, never
payloads that may contain business data or credentials.
- A /health endpoint.

### E. Tests
- Unit tests (Vitest or Jest) with mocked HTTP (for example nock or msw) for both APIs: token fetch and
caching, run trigger, template run, polling until terminal state, status mapping, defect creation,
retry/backoff and idempotency.
- Contract tests that load example payloads from the official specs into fixtures, so a schema change
breaks a test rather than production.
- No test may call a real tenant.

### F. Deployment and docs
- For Cloud Foundry: manifest.yml and an MTA descriptor (mta.yaml) that binds the credential store or
destination service; deployment with cf push or the MTA build tool. For Kyma: a Deployment, Service and
APIRule, with secrets from a Kubernetes Secret or the credential store.
- A README covering: architecture, prerequisites, creating the SAP Cloud ALM API service instance and key,
storing ACCELQ credentials, registering the endpoint in Cloud ALM Landscape Management (service type Test
Automation, role Test, endpoint with Basic or OAuth2 client credentials), the mapping file, running tests,
deploying and rolling back, and operations (logs, alerts, credential rotation).

## How to work
- Show me a plan and the file tree first and wait for my approval.
- Build in small steps and run the tests after each step.
```

```
- Wherever an endpoint, field or status value cannot be confirmed against the official API references
above, leave a clearly marked TODO with the doc link. Do not invent endpoints.
- Stop and ask me before any deployment, any call to a real tenant, or any change outside this repository.
```

> ⚠ **Review before you deploy**
>
> Generated code is a starting point, not a release. Read every file, check each endpoint and field against the official references, and run it against non-production Cloud ALM and ACCELQ tenants before anything touches production.

Pitfalls

- **Two masters for one test case.** If people edit automated test cases in both tools, the next sync overwrites someone. Design in the tool, plan and trace in Cloud ALM.
- **IDs that drift.** Renaming or cloning scenarios breaks mappings that rely on names. Map on IDs and version the mapping file.
- **Polling too hard.** Cloud ALM polls the provider and the adapter polls ACCELQ. Tight loops on both sides add load and burn API volume against fair-use limits [35]. Back off.
- **Stuck runs.** A job waiting for an agent that never comes stays queued [36]. Set a maximum wait and report it as an error so the test plan doesn't show it as running forever.
- **Defect floods.** One broken login step can fail 200 test cases. Deduplicate by root cause or let testers create defects after triage.
- **Credentials in the wrong place.** Keys in CI variables, laptops or repositories are the most common way integrations leak. Use the credential store and rotate.
- **Licensing assumptions.** Check that API access, execution agents and parallel runs are covered by your automation tool license, and that Cloud ALM API use stays within your entitlement [4].
- **Treating beta APIs as stable.** The Test Plans API is marked beta [14]. Isolate calls to it behind one module.
- **Porting the old suite.** SAP expects new test cases where UIs changed [2]. Rebuild around the Cloud ALM process scope, starting with what to automate first.

A 30-day rollout plan

DAYS 1–7: DECIDE AND PREPARE

- Agree test case ownership, ID mapping rules and defect policy with the test manager and release manager
- Confirm with your tool vendor whether a native connector exists; if it does, test it first
- Create the Cloud ALM API service instance (least-privilege scopes) and an ACCELQ integration user
- Pick one non-production Cloud ALM project and one process scope with five to ten automated scenarios

DAYS 8–15: BUILD AND CONNECT

- Build the adapter with unit and contract tests; deploy to a non-production BTP space
- Register the Test Automation service and endpoint in Landscape Management
- Synchronize the pilot scope and review test case names and data variants with a functional tester

DAYS 16–23: RUN AND HARDEN

- Execute the pilot test plan; verify statuses, logs and defects end to end
- Run the integration test checklist, including duplicate execute, agent loss and credential expiry
- Add monitoring, alerts and a runbook for the adapter

DAYS 24–30: PROVE VALUE AND DECIDE

- Run one real regression cycle through the integration alongside the old reporting

- Show the traceability and test execution analysis views to the steering committee
- Record the go/no-go on wider rollout in a Cloud ALM quality gate, with owners for scale-out

Frequently asked questions

Does Cloud ALM run automated tests itself?

No. Cloud ALM orchestrates and reports; providers that implement the Test Automation API execute [5],[1].

Who hosts the Test Automation API?

The provider. Cloud ALM calls the provider's endpoints, registered in Landscape Management as a Test Automation service [10],[6].

Does ACCELQ have a native Cloud ALM integration?

Not yet. ACCELQ's native SAP Cloud ALM integration is in development (roadmap per ACCELQ; not generally available), and it was not in ACCELQ's public documentation as of October 24, 2026 [47]. ACCELQ's published REST APIs cover what an adapter needs today: run, template run, run status and scenario listing [36],[37],[39],[40]. Ask ACCELQ for the current status before building.

Can I use this design with a different automation tool?

Yes. The provider side and Cloud ALM side are identical for every tool; only the client calls in the adapter change. Other registered providers follow the same contract [5].

Should automated failures create defects automatically?

Only with deduplication and an agreed policy. The documented path is creating defects from the test run after triage [25], which is a sensible default for the first cycles.

What about manual test cases built outside Cloud ALM?

Use the Test Management API, which creates manual test cases with activities and actions in one call [18]. Our Cloud ALM guide covers the spreadsheet route too.

Is there an API rate limit?

I could not find a published per-second limit for Cloud ALM APIs. Cloud ALM applies fair-use limits on API transfer volume per tenant [35], and ACCELQ caps page sizes (25 scenarios, 100 templates per page) [40],[38]. Build in backoff for 429 responses on both sides.

Summary and next steps

Cloud ALM is where SAP programs plan, trace and decide. Automated evidence only counts there if it arrives without anyone retyping it. The short version of this guide:

1. **Cloud ALM orchestrates; the provider executes.** Your tool, or an adapter in front of it, implements the Test Automation API and Cloud ALM calls it.
2. **Make three decisions before any code:** who owns the test case master, how IDs map, and who creates defects.
3. **Use a supported connector when one exists;** otherwise build a small, tested adapter on BTP from the documented APIs, with secrets in the credential store.
4. **Prove it on one scope in a non-production tenant,** then run a real regression cycle through it before you scale.

Two ways to move faster from here:

TRY THE WORKED EXAMPLE

Prove the automation side first

Start a free ACCELQ trial, automate five scenarios from one Cloud ALM process scope, and run them through the API calls in this guide. You will know within a week whether the integration has results worth moving.

Start the free ACCELQ trial

Go deeper: the Academy courses on test preparation and test plans in SAP Cloud ALM and transitioning from Solution Manager to SAP Cloud ALM, and RISE with SAP: why testing is still your job.

Sources

[1] Test automation integration with SAP's ALM platforms (partner information), SAP Support Portal, accessed October 2026

[2] Transition to SAP Cloud ALM for Implementation, SAP Support Portal, accessed October 2026

[3] Transition from SAP Solution Manager to SAP Cloud ALM, SAP Support Portal, accessed October 2026

[4] SAP Solution Manager and SAP Cloud ALM FAQ, SAP Support Portal, accessed October 2026

[5] Integrating Test Automation Providers, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[6] Integrating Other Test Automation Providers, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[7] Synchronizing Automated Test Cases, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[8] Executing Automated Test Cases, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[9] Automated Test Cases, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[10] APIs in SAP Cloud ALM, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[11] SAP Cloud ALM Test Automation API (CALM_TEST_AUTOMATION), version 1.0.13, SAP Business Accelerator Hub, accessed October 2026

[12] SAP Cloud ALM API package, SAP Business Accelerator Hub, accessed October 2026

[13] SAP Cloud ALM Test Management API (CALM_TM), SAP Business Accelerator Hub, accessed October 2026

[14] SAP Cloud ALM Test Plans API (CALM_TM_PLAN, beta), SAP Business Accelerator Hub, accessed October 2026

[15] SAP Cloud ALM Tasks API (CALM_TKM), SAP Business Accelerator Hub, accessed October 2026

[16] SAP Cloud ALM Analytics API (CALM_ANALYTICS_ODATA), SAP Business Accelerator Hub, accessed October 2026

[17] API examples for SAP Cloud ALM (SAP-samples/cloud-alm-api-examples), SAP on GitHub, accessed October 2026

[18] Showcasing the SAP Cloud ALM Test Management API (Jupyter notebook), SAP on GitHub, accessed October 2026

[19] SAP Cloud ALM Projects and Tasks API (Jupyter notebook), SAP on GitHub, accessed October 2026

[20] SAP Cloud ALM Features API with references (Jupyter notebook, task type codes), SAP on GitHub, accessed October 2026

[21] SAP Cloud ALM – Extend with APIs, Part 3: Work with SAP Cloud ALM API in the SAP API Hub, SAP Community, August 26, 2021

[22] Obtain a service key from a Cloud ALM API instance, New Relic documentation (third-party setup guide), accessed October 2026

[23] ALM plug-in for Grafana (Cloud ALM Analytics API data source), SAP on GitHub, accessed October 2026

[24] Test Execution Concepts, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[25] Creating Defects from a Test Run, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[26] Defects, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[27] Test Plans, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[28] Test Preparation Concepts, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[29] Requirement Traceability, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[30] Solution Process Traceability, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[31] Test Execution Analysis, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[32] Quality Gates, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[33] Status Flow of Features, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[34] Generating Manual Test Cases with Joule, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[35] Fair Use, SAP Help Portal (SAP Cloud ALM), accessed October 2026

[36] Run test job (POST .../awb/api/2.0/{tenantCode}/{projectCode}/test-exec/jobs/run), ACCELQ API documentation, accessed October 2026

[37] Trigger Run from Job template, ACCELQ API documentation, accessed October 2026

[38] List Job templates, ACCELQ API documentation, accessed October 2026

[39] Get test job run status (GET .../awb/api/1.0/{tenantCode}/runs/{runPid}), ACCELQ API documentation, accessed October 2026

[40] Fetch Scenario details, ACCELQ API documentation, accessed October 2026

[41] Run test job (ACCELQ Support, updated October 10, 2026), ACCELQ Support, accessed October 2026

[42] ACCELQ CI plugin for GitHub Actions (source), ACCELQ on GitHub, accessed October 2026

[43] ACCELQ CI Connect plugin for Jenkins (source), Jenkins project on GitHub, accessed October 2026

[44] Using Dockerized Agents for ACCELQ test execution, ACCELQ Support, accessed October 2026

[45] SAP Test Discovery & Automation, ACCELQ, accessed October 2026

[46] Setting up traceability between requirements and scenarios (updated July 15, 2026), ACCELQ Support, accessed October 2026

[47] Establishing integration with requirements and defect tracking tools (updated August 18, 2026), ACCELQ Support, accessed October 2026

[48] Running manual tests (updated August 22, 2024), ACCELQ Support, accessed October 2026

[49] How do you link manual and automated test cases? (updated August 22, 2024), ACCELQ Support, accessed October 2026

[50] Multi-modal automation (updated October 2, 2026), ACCELQ Support, accessed October 2026

[51] Support for API testing (updated November 4, 2025), ACCELQ Support, accessed October 2026

[52] Working with test cases in ACCELQ (updated October 12, 2026), ACCELQ Support, accessed October 2026

[53] Connect AI assistants to ACCELQ MCP (updated September 27, 2026), ACCELQ Support, accessed October 2026

[54] Self healing (updated August 29, 2026), ACCELQ Support, accessed October 2026

[55] Test suites in ACCELQ (updated June 9, 2025), ACCELQ Support, accessed October 2026

[56] SAP GUI automation with ACCELQ (updated January 16, 2026), ACCELQ Support, accessed October 2026

[57] ACCELQ Inc.: ACCELQ (partner listing; vendor-supplied text), SAP.com partner directory, accessed October 2026

ACCELQ

ACCELQ for SAP testing

Codeless, AI-powered test automation and test management for SAP and the applications around it. These are the capabilities that matter for the testing described in this guide.

Codeless SAP GUI automation

Automates SAP GUI transactions on ECC 6.0 and later (MM, FI, SD, CO, PP, HR) through transaction-code navigation, without scripting.

support.accelq.com/hc/en-us/articles/37423459738765

Web, Fiori, mobile and desktop in one flow

A recorder with smart element identification covers web apps (SAP Fiori is named in ACCELQ's SAP partner listing; pilot your own apps), iOS, Android and Windows desktop, mixed with API steps in a single scenario.

support.accelq.com/hc/en-us/articles/48792910342797

API, queue and file testing

Tests SOAP and REST services, message queues, service bus and FTP/file exchanges, with Swagger import and API mocking. This is how IDoc and EDI flows are checked end to end.

support.accelq.com/hc/en-us/articles/115007527587

Manual test management for business users

Business users run FUT, SIT and UAT steps in the Test Runner, mark results, attach screenshots and raise defects, alongside the automated suites.

support.accelq.com/hc/en-us/articles/360032862671

Requirement traceability

Links manual and automated tests to requirements and user stories in Jira, Azure DevOps and Rally, with requirement-based suites and coverage reporting.

support.accelq.com/hc/en-us/articles/360057668792

Data-driven, reusable tests

Test cases are data rows (with all-pairs generation) over reusable actions, contexts and an application universe, so one scenario covers many company codes and plants.

support.accelq.com/hc/en-us/articles/41589227364237

AI test generation and self-healing

Autopilot generates test cases from scenario logic, an MCP server lets AI assistants plan and run tests after you approve the plan, and self-healing keeps tests running as screens change.

support.accelq.com/hc/en-us/articles/49193518914445

File and data verification

Validates XML, fixed-width, CSV and PDF outputs, and checks values directly in Oracle, SQL Server, PostgreSQL, Snowflake and other JDBC databases.

support.accelq.com/hc/en-us/articles/360024031091

CI/CD and REST API

Triggers suites from Jenkins, GitHub Actions and Azure DevOps pipelines, and a documented REST API starts jobs and reads results.

accelq-docs.github.io/external-api-doc

Agent behind the firewall

A pull-only local agent runs tests against on-premise SAP systems with no inbound ports opened.

support.accelq.com/hc/en-us/articles/360019691372

COMING

Native SAP Cloud ALM integration In development (roadmap, per ACCELQ; not yet generally available): consume blueprint requirements and fit/gap decisions from SAP Cloud ALM, generate the matching TUT, FUT and SIT tests, and promote them to regression packs.

Start a free ACCELQ trial accelq.com/freetrial

ACCELQ

Codeless, AI-powered test automation for SAP and every enterprise application.

Start a free trial	accelq.com/freetrial
Learn more	www.accelq.com
Connect with Matt Angerer	www.linkedin.com/in/mattangerer
Talk to ACCELQ about your SAP program	accelq.com/contactus