FIELD MANUAL

The Clean Core Practitioner's Manual

An End-to-End Guide to ABAP Cloud, Extensibility Architecture and Custom Code Remediation

Release contracts and API state. The three-tier extensibility model. Package architecture, CDS and RAP extension patterns. Brownfield inventory, triage and the remediation pattern catalogue. AI-assisted triage and its failure modes. Governance, quality gates and metrics. A packaged client assessment and a twelve-month specialist path.

Prepared for Sayan Samanta
SAP Technology Lead
August 2026

Contents

How to Use This Manual

This is a working reference, not a textbook. It is written for someone who already knows ABAP well and needs the architecture, the decision rules and the remediation method — not an introduction to CDS or a tour of SE80.

It is organised in eight parts:

- **Part I — Foundations.** What clean core is, why it exists, the mechanics (release contracts, language versions, deployment models) everything rests on, and the **prerequisite and readiness check** (Chapter 5) you run before a programme starts.
- **Part II — The Decision Framework.** How to choose an extension approach and defend it, plus **Chapter 8**, the design lookup table for Private Cloud versus Public Cloud across every architectural concern.
- **Part III — Clean Core by Project Type.** The numbered rule set (Chapter 9), then how it binds in greenfield, brownfield conversion, selective transition, rollout, upgrade, carve-out, two-tier, BTP-led and run.
- **Part IV — Building Clean.** Patterns for greenfield and new development, including security and data protection (Chapter 17), integration architecture (Chapter 18), testing architecture (Chapter 19), and **Chapter 20, one requirement worked end to end through every decision in this book.**
- **Part V — Performance Engineering.** Compliance and performance are independent problems. The HANA performance model, the full checking checklist (Chapter 22), SQL and CDS tuning, ABAP-side cost, RAP and UI performance, and the measurement process.
- **Part VI — Remediation. Chapter 27 is the end-to-end migration runbook** — ten phases with entry and exit criteria, plus the per-object micro-process. Chapters 14–17 are the mechanics it references: inventory, triage, the remediation pattern catalogue, and where AI genuinely helps.
- **Part VII — Governance and Delivery.** Landscape, transport and lifecycle (Chapter 32), standards, quality gates, pipelines, metrics, and how to run a client assessment.
- **Part VIII — The Specialist Path.** A twelve-month plan and the traps.

Four entry points, depending on why you opened this. Starting a programme: Chapter 5, then Chapter 19. Scoping a project: Chapter 9, then the chapter for your project type. Designing a solution: Chapter 7, then Chapter 8. Reviewing code: Chapter 22, then Chapter 33.3. Seeing it all in one place: Chapter 16. Fixing one object: Chapter 27’s micro-process, then Chapter 22.

Read Parts I and II once, properly. Use Parts III, IV and V as lookup. Part VI is what turns individual skill into a repeatable service. Part VII is the career argument.

A warning about how this is used. Reading this makes you conversant. It does not make you a specialist. The credential in this field is a live conversion where you personally triaged several thousand objects and made calls that held up in production. Use this to structure that experience, not to substitute for it.

On version drift. SAP’s release contract catalogue, ATC check variants and Fiori app names change with each release. Everything technical here should be verified against the target system’s actual release before you present it to a client. The framework is stable; the object names are not.

Part I — Foundations

Chapter 1: What Clean Core Actually Is

1.1 The definition that matters

Clean core is not "don't write custom code." It is: **keep the standard software upgradable by ensuring every extension depends only on interfaces the vendor has contractually committed to keep stable.**

Everything else — the tiers, the release contracts, the language version enforcement — is machinery serving that one sentence.

1.2 The five dimensions

SAP frames clean core across five dimensions. Most practitioners only work on one and then claim the whole thing.

Dimension	What "clean" means	Typical failure
Extensions	Custom code uses released APIs and registered extension points only	Direct table reads, unreleased FM calls, modifications
Data	Master data is governed, deduplicated, consistent	Six customer master variants, no golden record
Processes	Business processes follow standard flows where possible	Process forked to preserve a 1998 workaround
Integrations	API-first, event-driven, governed through a hub	Two hundred point-to-point RFC/IDoc interfaces
Operations	Automated, monitored, transparent lifecycle	Manual transports, no test automation

The practitioner's point: a system can be 100% ABAP Cloud compliant and still be a filthy core through master data chaos and point-to-point integration. When a client says "we're doing clean core," ask which dimension. Nine times out of ten they mean extensions only. That gap is where a good consultant adds visible value in the first meeting.

1.3 Why SAP needs it — the honest version

Three reasons, in descending order of technical legitimacy.

1. Multi-tenancy is impossible without stable interfaces. This is the real constraint. In S/4HANA Cloud Public Edition, thousands of tenants share a codebase that SAP upgrades quarterly. If tenant code reads `VBAK` directly or calls an unreleased function module, SAP can never refactor either. The set of "things we can never change" grows until the product freezes. Release contracts make that surface finite and testable.

2. Upgrade economics. In the ECC era every installation was a snowflake. SAP maintained thousands of divergent systems; the upgrade cost landed on the customer; customers responded by not upgrading for a decade. Both sides lost.

3. Commercial. Subscription revenue is valued more highly than perpetual licence plus maintenance, and clean core lowers SAP's cost of moving customers onto it. The 2027 ECC maintenance deadline is the forcing function.

All three are true simultaneously. Reason 1 is sound engineering and you should argue it with conviction. Reason 3 is real and pretending otherwise damages your credibility with sceptical clients, who have usually noticed. The correct posture with a client is: *the constraint is genuine, the benefit is concentrated in upgrade cost and future optionality, and the deadline is doing more of the persuading than the architecture is.*

1.4 What clean core buys the customer

Be specific. Vague benefit claims are why clean core programmes fail to get funded.

- **Upgrade effort approaching zero for compliant code.** No SPAU/SPDD queue, no regression sweep across hundreds of Z-programs because SAP renamed a field. If the client has done an EHP upgrade, they can price this from memory.
- **Optionality on Public Cloud.** Compliant extensions are portable. Non- compliant ones lock the client to Private Cloud permanently.
- **Faster innovation adoption.** Embedded AI, new Fiori apps and analytics land without a remediation project first.
- **Lower total custom code volume.** Remediation projects routinely retire 40–60% of custom objects because nobody has executed them in years. That is a hard, measurable saving.

1.5 What it costs — say this out loud

- Released API coverage is **incomplete**. You will hit business logic with no C1 equivalent. Every experienced practitioner has.
- In Private Cloud you fall back to wrapper classes that exist only to satisfy a syntax checker. Real maintenance, zero business value.
- In Public Cloud there is **no fallback at all**. If it is not released, you redesign or file an influence request.
- Brownfield migration cost is front-loaded; the benefit accrues over future upgrades. This asymmetry is why "clean core as a big-bang programme" fails.

The defensible position for a brownfield client: **apply ABAP Cloud strictly to all new development starting now, freeze existing classic code as tier 3, and refactor only what you are touching anyway.** You capture most of the upgrade benefit without funding a migration with no direct payback.

Chapter 2: The Deployment Spectrum

Everything about what is permitted depends on where the system runs. Get this wrong and every downstream recommendation is wrong.

	On-Premise	Private Cloud (RISE)	Public Cloud
Tenancy	Single	Single	Multi
Upgrade control	Customer	Customer (within window)	SAP, quarterly
Classic ABAP	Yes	Yes	**No**
Modifications	Yes	Discouraged	**Impossible**
SE80 / SE38	Yes	Yes	**No**
Developer extensibility	Yes	Yes	Yes
Tier 2 wrappers	Yes	Yes	**No**
Direct DB table access	Yes	Yes	**No**
ABAP Cloud enforced	Optional	Optional	Mandatory

2.1 Where ABAP Cloud is genuinely mandatory

S/4HANA Cloud Public Edition and SAP BTP ABAP Environment. There is no other code path. The language version is enforced at syntax check. Nobody is choosing to comply; there is nothing else to do.

2.2 Where it is a recommendation

On-Premise and Private Cloud Edition. You retain full classic ABAP. SAP strongly advises ABAP Cloud. Nothing enforces it unless you enforce it.

Anyone telling a Private Cloud client that ABAP Cloud is “required” is repeating positioning as if it were a technical constraint. That is a credibility loss when the client’s own developer demonstrates otherwise in ten seconds. Say “recommended, and here is what it costs you not to.”

2.3 The practical consequence for your work

Most brownfield conversions land in **Private Cloud**, not Public. Complex discrete manufacturing, heavy variant configuration and industry-specific processes frequently do not fit Public Cloud, which is precisely why Private Cloud exists. That means:

- Tier 2 wrappers are available to you and you will use them.
- The enforcement is organisational, not technical — governance matters more than tooling.
- Your job is largely *voluntary* compliance, which is harder to sell and harder to sustain than mandatory compliance.

Chapter 3: Release Contracts and API State

This is the mechanical core of clean core. Know it cold.

3.1 The contracts

Contract	Name	Meaning
C0	Extension point	You may *extend* the object at registered hooks. SAP guarantees stability of the extension point.
C1	Stable interface / system-internal use	You may *consume* the object from other ABAP code. This is the contract that governs language version access.
C2	Remote API	Stable for *external* consumers over OData, SOAP or RFC.
C3	Business configuration	Stable for configuration content.
C4	AMDP / database interface	Stable for use in AMDP methods.

3.2 Visibility flags — the part that confuses everyone

For **C0 and C1 only**, SAP additionally sets a visibility. That produces the four labels you actually see in ADT:

Label	Contract	Grants
Use in Cloud Development	C1	Consume from ABAP for Cloud Development code
Extend in Cloud Development	C0	Extend at registered extension points from cloud code
Use in Key User Apps	C1	Consume from Custom CDS Views / Custom Logic
Extend in Key User Apps	C0	Field extensions via key user Fiori apps

An object can carry several of these, or one, or none. **Use does not imply Extend and Extend does not imply Use.** Plenty of SAP views are C1-consumable and not extensible; some extension points sit on objects you cannot freely consume.

3.3 The four traps

Trap 1 — C2 is not the same family. C2 is stability for *external* consumers. Finding a service on SAP Business Accelerator Hub tells you nothing about whether you can `SELECT`

from the underlying entity in on-stack ABAP Cloud code. Different contract, different question. This is the single most common misunderstanding in the field.

Trap 2 — release state is not portable across products. An object C1- released in S/4HANA on-premise 2023 may not be released in Public Cloud, and the reverse happens. Always verify in the *target* system.

Trap 3 — release state is not portable across releases. Objects get released over time; a few get deprecated. Check against the client's actual release, not your sandbox.

Trap 4 — "Extend" does not mean "extend anywhere." C0 guarantees stability only at dedicated, registered extension points. You extend where SAP put a hook, not where the field would be convenient.

3.4 How to find released objects

In descending order of usefulness:

1. **ADT → Released Objects node** in the project tree. Browsable, filterable, authoritative for that system.
2. **ADT → ABAP Repository Tree** with a property filter, e.g. `api:use_in_cloud_development` or `api:extend_in_cloud_deve*`. Best when you know roughly what you want.
3. **Properties → API State tab** on any individual object. Use this to confirm before you build.
4. **SAP Business Accelerator Hub** — remote APIs (C2) only. Useful for integration design, misleading for on-stack development.

Build the habit of checking API State *before* designing, not after the syntax check fails.

3.5 Deprecation

Released objects can be marked deprecated with a successor. Deprecated does not mean broken; it means plan the move. In a remediation project, treat deprecated dependencies as findings even though they compile — they are next year's findings otherwise.

Chapter 4: ABAP Language Versions

Three language versions exist. The version is a property of the **package** (via its software component), and it determines what the syntax check permits.

Version	Where	Access
Standard ABAP	On-prem, Private Cloud	Everything. No restrictions.
ABAP for Cloud Development	Public Cloud, BTP ABAP Env, and *optionally* on-prem/Private Cloud	C1-released objects only. Restricted language subset.
ABAP for Key Users	Key user extensibility tools	Very narrow subset, restricted editor

4.1 What ABAP for Cloud Development removes

Not an exhaustive list, but the ones that bite in remediation:

- No direct access to non-released DDIC tables — every `SELECT ... FROM VBAK` becomes a finding
- No `CALL FUNCTION` to unreleased function modules
- No classic Dynpro, no SAP GUI-based UI, no `WRITE` list output
- No `SUBMIT`, no `CALL TRANSACTION`
- No native SQL, no `EXEC SQL`
- No file system access via `OPEN DATASET`; use released APIs instead
- No `SYSTEM-CALL`, no kernel calls
- Obsolete language constructs blocked (short forms, `MOVE`, header lines, `TABLES`, implicit conversions in many cases)
- Statements requiring authority checks are constrained to released patterns

4.2 The single most important configuration decision

Set the ABAP language version at package creation. It is painful to change afterwards.

Concretely, when you create a package for cloud-compliant development:

- Assign it to software component `ZLOCAL` (or a customer software component you have created), and
- Ensure the ABAP language version resolves to *ABAP for Cloud Development*.

If you create a plain Z package under `HOME` in an on-premise system, you get Standard ABAP and full access to everything. Your code will compile happily and then be rejected in Public Cloud. Teams have written entire modules this way and discovered it at the wrong moment.

4.3 Using cloud language version on-premise deliberately

On-premise and Private Cloud let you *opt in* to ABAP for Cloud Development. Do this for all new development. It gives you:

- Compile-time enforcement instead of code review enforcement
- Immediate, unambiguous findings rather than an ATC report nobody reads
- Genuine portability of the resulting objects

This is the highest-leverage governance decision available on a brownfield programme, and it costs nothing but discipline.

Chapter 5: Prerequisite and Readiness Check

Run this before the programme starts. Every item that is missing becomes a delay later, and most of them have lead times measured in weeks — access requests, licence procurement, and above all usage data, which cannot be back-filled.

The rule: do not start analysis until section 5.2 is green. Do not start remediation until 5.3, 5.4 and 5.5 are green.

5.1 Scoping prerequisites — establish before anything else

These are questions, not tasks. If you cannot answer them, you cannot scope.

Question	Why it decides the whole approach
Target deployment model — On-prem, Private Cloud, Public Cloud?	Determines whether tier 2 exists at all. Every recommendation changes.
Conversion type — brownfield, greenfield, selective data transition?	Greenfield means no remediation, only architecture.
Target S/4HANA release and planned upgrade cadence?	Release state of APIs differs by release.
Is Public Cloud a future option, or definitively ruled out?	Decides whether tier 2 is a bridge or a permanent resting place.
Who owns the decision to delete a custom object?	If nobody, the retire wave stalls and the business case collapses.
Is there an existing custom code governance process?	Determines whether you are building or repairing governance.

The most common scoping error: assuming Public Cloud constraints on a Private Cloud client. It makes you look rigorous and produces an over-engineered, over-priced recommendation. Confirm the target in writing.

5.2 Technical prerequisites for analysis

Item	Requirement	Lead time
Usage logging (UPL / SCMON)	Active in **production** for a minimum 12-month window covering all period-end cycles	**Cannot be back-filled.** Turn on immediately, even if the programme starts later.
Central ATC check system	Release **equal to or higher** than the systems being checked. A lower-release check system cannot evaluate higher-release code.	Days to weeks
RFC destinations	Check system → each checked system, with the right authorisations	Days
ADT	Current version, matching a supported Eclipse release	Hours
Object inventory extract	Full repository listing with package, type, author, last-changed	Hours
Where-used data	Cross-reference index built and current	Hours to days
Read access to production	For usage extraction; read-only is sufficient and easier to obtain	Days to weeks

The usage logging point deserves emphasis. Without it you are triaging on guesswork, and the retire wave — the phase that funds the programme — becomes undefendable. If a client engages you with no usage data, your first recommendation is to enable it and revisit in twelve months, or to accept a materially weaker retire case. Say this in week one, not month three.

5.3 Technical prerequisites for remediation

Item	Requirement
Development system	Target release, cloud-ready, with the agreed package structure created
Test system	Production-like **data volume and shape**, not an empty shell
Data refresh process	Defined cadence and a client-approved anonymisation approach
abapGit	Installed, with a repository strategy agreed
Transport route	Defined, with the ATC gate configured on release
ATC check variants	`Z_CLOUD_STRICT` and `Z_REMEDIATION` created and agreed
ABAP Unit baseline	Existing coverage measured; near-zero is normal and must be planned for
BTP subaccount	Only if side-by-side is in scope — entitlements, Cloud Connector, identity setup

5.4 Organisational prerequisites

Technical readiness without these produces a stalled programme.

- **A named business owner per functional area**, with authority to approve retirement. Not a nominee — someone who will actually attend workshops.
- **An architect with authority to refuse an exception.** If exceptions are granted by whoever is under deadline pressure, governance does not exist.
- **A decision forum** with a fixed cadence for dispositions that need escalation. Weekly during triage.
- **Testing capacity identified.** Testing is 40–60% of remediation effort. A programme that has not resourced it will slip, and you should say so before it starts.
- **A change freeze policy** for objects marked FREEZE, and a process for handling a business request to change one anyway.

5.5 Data and environment prerequisites

- Production-like test data covering edge cases: reversals, cancellations, multiple currencies, multiple company codes, period-end states.
- A defined method for **semantic equivalence testing** — how you will compare old and new result sets. Usually a comparison harness that runs both paths and diffs. Build this once, use it on every object.
- Agreement on what constitutes an acceptable difference. Rounding, sort order, and null-versus-blank will all differ; decide in advance which matter.

5.6 The go / no-go checklist

Before analysis begins:

- ☐ Target deployment model confirmed in writing
- ☐ Target release confirmed
- ☐ Usage logging active, window length known and stated
- ☐ Central ATC system at or above target release
- ☐ RFC connectivity tested end to end
- ☐ Object inventory extracted
- ☐ Business owners named per functional area
- ☐ Architect with exception authority identified

Before remediation begins:

- ☐ Package structure and language version policy created
- ☐ Check variants agreed and the transport gate active
- ☐ Test system with production-like data available
- ☐ Semantic equivalence harness built and proven on one object
- ☐ Testing capacity resourced
- ☐ Disposition register signed off for the objects in this wave
- ☐ Tier 2 wrapper policy agreed, with a cap

Any unticked box is a risk register entry with a named owner and a date. Do not proceed by hoping it resolves itself.

Part II — The Decision Framework

Chapter 6: The Three-Tier Extensibility Model

SAP's tier model is the single most useful frame for talking to architects. Learn to draw it on a whiteboard from memory.

6.1 The tiers

Tier 1 — Developer Extensibility (cloud-compliant, on-stack). ABAP for Cloud Development, using only C1-released APIs and C0 extension points. Upgrade-stable. Available in all deployment models. **This is the target state for every new object.**

Tier 2 — Cloud API Enablement (wrapper layer). Classic ABAP objects that you write and then *release yourself* with a C1 contract, so tier 1 code may consume them. Used when SAP has not released something you genuinely need. *Available in on-premise and Private Cloud only. Not in Public Cloud.*

Tier 3 — Classic Extensions. Unrestricted classic ABAP. Modifications, direct table access, unreleased API calls. Not upgrade-stable. This is where the existing brownfield estate lives.

6.2 The critical framing

Tiers 2 and 3 are **containment strategies, not destinations.** The purpose of tier 2 is to concentrate all non-compliant dependency into a thin, inventoried, deliberately-owned layer, so that:

- The blast radius of an SAP change is a known list of wrapper objects
- Your business logic in tier 1 remains portable
- When SAP eventually releases the real API, you change one wrapper and nothing else

A tier 2 layer that grows without governance is just tier 3 with extra steps. Cap it, inventory it, review it quarterly.

6.3 The tier 2 wrapper pattern

The mechanics, in order:

1. Create a classic ABAP class in a **Standard ABAP** package, in a clearly named wrapper namespace (e.g. `ZCC_WRAP_*`).
2. Implement the minimum surface needed. Do **not** pass through raw SAP structures — define your own DDIC types or ABAP types at the boundary, so an SAP structure change does not propagate into tier 1.
3. Keep it thin: no business logic in the wrapper. Fetch, convert, return.
4. Release the wrapper with a C1 contract (ADT → object → *Release API*, or the API State tab).
5. Consume it from your ABAP Cloud package.

Rules for the wrapper layer:

- One wrapper per SAP capability, not one per consumer
- Document *why* the wrapper exists and what released API would replace it
- Add a review date. Check whether SAP has released the real thing.
- Never let a wrapper leak an unreleased type into its signature

6.4 A worked example of the decision

Requirement: on saving a sales order, validate the customer's credit against a figure held in a legacy Z-table, and block the save if exceeded.

- Is there a released API to read the Z-table? It is a custom table — you own it, so you release it yourself or expose it via a released CDS view. No wrapper needed.
- Is there a released extension point on the sales order save? Check the BAdI catalogue and the API State of the relevant enhancement spot. If yes, tier 1.
- If the only hook is an unreleased classic BAdI, you are in tier 2: implement the classic BAdI, and have it delegate immediately to a tier 1 class that holds the actual logic.

That delegation pattern is the single most valuable habit in this book. Put the logic in cloud-compliant code and let the non-compliant hook be a two-line call. When the hook is eventually released, you delete two lines. When it is not, your logic is still portable and testable.

Chapter 7: Choosing an Extension Approach

7.1 The three styles

	Key User	Developer (on-stack)	Side-by-Side (BTP)
Runs	On stack	On stack	On BTP
Built by	Business power user	ABAP developer	Full-stack / ABAP
Tooling	Fiori apps	ADT	ADT / CAP / BAS
Lifecycle	Coupled to core	Coupled to core	Independent
Latency to core data	None	None	Network
Scales independently	No	No	Yes
Skills needed	Low	Medium	High

7.2 The decision sequence

Work down this list and stop at the first "yes." This ordering is deliberate: each step down adds cost, complexity and skill requirement.

1. **Can standard configuration do it?** Exhaust this first. A surprising proportion of "requirements" are unconfigured standard.

2. **Can key user extensibility do it?** Custom fields, custom logic, custom CDS views, custom business objects, adaptations to Fiori apps. No developer involvement, no transport of code.
3. **Can on-stack developer extensibility do it?** RAP, CDS, released BAdIs, ABAP Cloud. Choose this for anything that needs core data with no latency, or transactional consistency with the core.
4. **Does it need to run separately?** Side-by-side on BTP.

7.3 When side-by-side is genuinely right

Side-by-side is oversold as the “modern” default. It is right when:

- The extension needs to scale independently of the ERP
- It integrates several systems, not just S/4HANA
- It needs a technology the ABAP stack does not offer (specific ML runtime, particular UI framework, high-volume event processing)
- It must survive an S/4HANA re-implementation
- The consumer is external (partners, customers, public)
- The release cadence must be independent of the core’s transport calendar

7.4 When side-by-side is the wrong answer

- Field-level extensions to core objects
- Validations and determinations on save
- Anything needing tight transactional consistency with the core
- Anything a Fiori elements app over a projection view would solve
- Anything where the round-trip latency is user-visible

For these, on-stack developer extensibility is objectively better engineering. SAP’s own decision guidance agrees; the marketing does not. Side-by-side buys you latency, data replication, token and auth complexity, two lifecycles, two skill sets and two operational surfaces. Charge for that only when it is worth it.

7.5 The costs nobody prices in

When you recommend side-by-side, price these explicitly or you will be blamed later:

- BTP consumption cost, which is variable and hard to forecast
- Identity propagation and principal propagation setup
- Cloud Connector for on-premise reachability
- A second CI/CD pipeline and a second monitoring surface
- Data replication and the consistency questions it opens
- A second on-call rotation

Chapter 8: Architecting by Deployment Model

The same requirement gets a different design depending on where the system runs. This chapter is the lookup table for that. Use it in design workshops.

Throughout: **PCE** = Private Cloud Edition (and on-premise, which behaves the same for these purposes); **Public** = S/4HANA Cloud Public Edition.

8.1 The governing difference

In PCE you *choose* to be clean. In Public you *are* clean because nothing else compiles. Everything below follows from that.

The second-order consequence matters more: in PCE, every non-compliant choice is a decision someone made and can be traced, priced and reversed. Your job is to make those decisions visible and deliberate rather than accidental.

8.2 User interface

Requirement	PCE	Public
Transactional app	Fiori elements over RAP. Classic Dynpro possible but tier 3.	Fiori elements over RAP. Only option.
Complex/freeform UI	SAPUI5 freestyle over an OData V4 service	Same
Reporting UI	Fiori elements analytical list page over a CDS analytical query	Same
Quick internal tool	Temptation: an ALV report. Cost: tier 3 forever.	Not available
Adaptation of a standard app	UI Adaptation (key user)	UI Adaptation (key user)

Architecture note: the PCE “quick ALV report” is the single largest source of new tier 3 debt on live programmes. It is fast, familiar, and it compounds. Ban it in the standards document or expect to remediate it later.

8.3 Business logic and persistence

Requirement	PCE	Public
New custom business object	RAP managed BO in a cloud package	Same
Custom persistence	Custom tables, released with C1, exposed via a CDS interface view	Same
Logic on save of an SAP object	Released BAdI or RAP behaviour extension. Fallback: classic BAdI delegating to a tier 1 class.	Released BAdI or behaviour extension only. No fallback.
Reading SAP data	Released CDS view. Fallback: tier 2 wrapper.	Released CDS view only.
Calling SAP logic	Released class or BAPI. Fallback: tier 2 wrapper.	Released API only.
Complex calculation	ABAP Cloud class. AMDP only over C4-released artefacts.	Same

8.4 Integration

Requirement	PCE	Public
Inbound synchronous	Released OData/RAP service, or a custom service binding	Communication scenario + communication arrangement
Outbound synchronous	Released outbound API, HTTP client via released classes	Communication arrangement, outbound service
Asynchronous / event	RAP business events, Event Mesh / Advanced Event Mesh	Same
Legacy IDoc	Available. Tier 3 unless the IDoc interface is released.	Very limited. Redesign to API or event.
Legacy RFC inbound	Available, tier 3	**Not available.** Redesign mandatory.
File-based interface	`OPEN DATASET` available but tier 3. Prefer an integration layer.	**No filesystem.** Integration Suite or equivalent, mandatory.
Middleware	Optional	Effectively mandatory for anything non-trivial

The migration pattern that matters: PCE clients with two hundred point-to-point RFC and file interfaces cannot move to Public without rebuilding the integration layer entirely. When a client asks “what stops us going Public,” integration is usually a bigger blocker than custom code — and almost nobody scopes it. Raising this is a strong differentiator in an assessment.

8.5 Output, forms and printing

Requirement	PCE	Public
Form output	Adobe Forms via released output management	Output management + Adobe Forms, key user configurable
Legacy SAPscript / SmartForms	Available, tier 3	Not available
Custom output determination	BRF+ / output management, or released BAdIs	Output management only
Email	Released mail API	Released mail API

8.6 Background processing

Requirement	PCE	Public
Scheduled job	Application Jobs framework with a job catalogue entry, ideally. Classic SM36/SM37 available but tier 3.	Application Jobs only
Job class / template	RAP-based job catalogue entry implementing the job interface	Same
Parallel processing	Released parallelisation APIs	Same
`SUBMIT` of a report	Available, tier 3	Not available

Design implication: every background process must be expressible as a job catalogue entry with typed parameters. This is a genuine constraint on legacy reports that take a selection screen with forty fields.

8.7 Authorisations

Requirement	PCE	Public
Custom authorisation object	Available	Available, with restrictions
Authority check	Released authorisation APIs; `AUTHORITY-CHECK` restricted in cloud language version	Same
Role design	PFCG	Business roles / business catalogues, key user maintained
Restriction on RAP BO	Instance-based authorisation in behaviour definition	Same

8.8 Extensions to SAP objects

Requirement	PCE	Public
Add a field to an SAP table	Extension include / append structure, plus key user custom field	Key user custom field only
Add a field to an SAP CDS view	`extend view entity`, if C0 released	Same, plus key user field publication
Add a field to an SAP Fiori app	Custom field published to the UI, or UI adaptation	Same
Modify SAP code	Technically possible. Never.	Impossible
Implicit enhancement	Available, tier 3	Not available
Explicit enhancement point	Available, tier 3 unless released	Not available
Classic BAdI	Available; tier 2 or 3 depending on release state	Released BAdIs only

8.9 Analytics

Requirement	PCE	Public
Operational report	CDS analytical query over released cubes	Same
Custom KPI	Custom analytical query (key user) or CDS query	Same
Data extraction to a warehouse	CDS-based extraction, released extractors	CDS-based extraction only
Direct DB access from BI	Possible, discouraged	Not possible

8.10 Designing for portability

If Public Cloud is a live future option — and for most clients it should at least be an option rather than a foreclosed one — design PCE extensions so they could move. Concretely:

1. **All business logic in ABAP for Cloud Development packages.** No exceptions.
2. **Non-compliant dependency isolated in `ZCC_WRAP` only**, with the register from Appendix C maintained.
3. **No filesystem or RFC dependency in business logic** — route through an interface abstraction even where the underlying transport is legacy.
4. **No Dynpro.** UI is Fiori elements or freestyle SAPUI5, always.
5. **Jobs modelled as job catalogue entries**, even where SM36 would work.
6. **Integration through an abstraction layer**, so the transport can be swapped without touching logic.

Then track one metric: **objects blocking a Public Cloud move.** That number, trending toward zero, is the clearest expression of clean core progress an executive will understand, and it is the number that justifies the programme when someone asks what the money bought.

8.11 When to deliberately break the rules

Clean core is a means. Break it knowingly when:

- The client has definitively ruled out Public Cloud and the remediation cost exceeds the lifetime upgrade cost of the object
- A regulatory deadline makes the compliant design infeasible in the time available
- The object is scheduled for retirement within the next upgrade cycle

In all three cases: record the decision, name the owner, set a review date, and put it in the exception register. **An undocumented deliberate exception is indistinguishable from negligence eighteen months later**, and you will be the one asked to explain it.

Part III — Clean Core by Project Type

The rules are constant. Where they bind, what they cost and who enforces them changes completely with the shape of the project. This part is the bridge between principle and delivery.

Chapter 9: The Universal Rule Set

Numbered so they can be cited. Put these in the client's development standards verbatim, with the rule IDs intact, so a code review comment reads "violates CC-D3" rather than "I don't like this."

9.1 How to read the rule set

Each rule carries a **class**:

- **Absolute** — no exception, ever. Granting one destroys the category.
- **Conditional** — exception permitted through the register, with an owner and a review date.
- **Contextual** — binds only in certain deployment models or project types.

And an **enforcement level**. Enforce every rule as far left as possible:

```
Compiler  >  Gate  >  Review  >  Policy
(cannot break)  (cannot ship)  (caught)  (hoped for)
```

A rule enforced only by policy is a rule that will be broken under deadline pressure. If you cannot move a rule left, expect to see it violated and plan for that rather than being surprised.

9.2 Architecture rules

ID	Rule	Class	Enforced by
CC-A1	No modification of SAP objects	Absolute	Review + policy
CC-A2	Extensions depend only on released APIs and registered extension points	Conditional	Compiler (cloud language version)
CC-A3	Deployment model and target release confirmed in writing before design	Absolute	Gate (project)
CC-A4	Extension approach chosen by the decision sequence, with the rejected options recorded	Absolute	Review
CC-A5	Business logic never lives in a non-compliant hook — the hook delegates	Conditional	Review
CC-A6	Integration is API- or event-based; no database-level integration	Absolute	Review + policy

9.3 Development rules

ID	Rule	Class	Enforced by
CC-D1	All new objects in ABAP for Cloud Development packages	Conditional	Compiler
CC-D2	Exactly one Standard ABAP package (`ZCC_WRAP`), architect-approved	Conditional	Review
CC-D3	Every wrapper has a justification, an owner and a replacement condition	Absolute	Gate (register)
CC-D4	No transport released with priority 1 or 2 ATC findings	Absolute	Gate (transport)
CC-D5	Data access via CDS interface views, never direct table select	Conditional	Compiler + review
CC-D6	UI annotations in metadata extensions, not inline	Conditional	Review
CC-D7	Unit tests exist and run without a populated system	Conditional	Gate (pipeline)
CC-D8	Volume assumption stated and tested at production-like volume	Conditional	Review

9.4 Governance rules

ID	Rule	Class	Enforced by
CC-G1	Exceptions granted only by the architect, in writing, with a review date	Absolute	Process
CC-G2	Exception register reviewed quarterly	Absolute	Process
CC-G3	Wrapper register reviewed against newly released APIs each release	Conditional	Process
CC-G4	Custom object count reported monthly	Conditional	Process
CC-G5	Gate ownership assigned to a named internal role before consultants leave	Absolute	Process

9.5 Data and integration rules

ID	Rule	Class	Enforced by
CC-I1	Every interface has a named owner	Absolute	Register
CC-I2	Asynchronous consumers are idempotent	Conditional	Review
CC-I3	Contracts versioned, with a deprecation policy	Conditional	Review
CC-I4	Personal data annotated on custom entities	Absolute	Review

9.6 Which rules bind where

Rule group	Public Cloud	Private Cloud / On-prem
CC-A2, CC-D1, CC-D5	Enforced by the platform	Voluntary — you must enforce them
CC-D2, CC-D3	Not applicable (no tier 2)	Critical
CC-A1	Impossible to violate	Must be policed
Everything else	Applies	Applies

The asymmetry that matters: in Public Cloud the platform does your governance. In Private Cloud you are the governance, and the failure mode is silent. This is why Private Cloud clean core is harder to sell and harder to sustain, and why the enforcement level in 9.1 is the most practical thing in this chapter.

9.7 Rule adoption sequence

You cannot switch on twenty rules on day one. Order for a client starting from nothing:

1. **CC-A3** — confirm the target. Everything else depends on it.
2. **CC-D1 and CC-D2** — package structure and language version. Moves the biggest rule to the compiler in one step.
3. **CC-D4** — the transport gate. Unavoidable, therefore effective.
4. **CC-G1** — the exception process, so rule 2 does not simply get ignored.
5. **CC-A1, CC-A5, CC-D3** — the architecture disciplines.
6. Everything else.

Steps 2 and 3 together deliver most of the value. A programme that does only those two, properly, beats one that publishes a thirty-page standard nobody enforces.

Chapter 10: New Implementation (Greenfield)

The cheapest place in the world to be clean, and most greenfield projects still fail at it. Understanding why is worth more than the rule list.

10.1 Why greenfield projects end up dirty

No legacy code, no remediation backlog, no conversion deadline — and yet the custom object count at go-live is routinely in the hundreds. The mechanism is always the same:

Fit-to-standard workshops produce gaps. Gaps produce pressure. Pressure produces extensions. The business describes the process it has today, the consultant records the delta from standard as a gap, and the gap becomes a development ticket. Nobody ever asked whether the process itself should change.

A greenfield implementation that re-creates the legacy system in S/4HANA is the single most expensive failure in this book, because the client paid greenfield prices for brownfield debt and has no remediation budget left.

10.2 Day-one decisions that are effectively irreversible

Make these before the first workshop, not after:

Decision	Why it is irreversible
Deployment model	Determines whether tier 2 exists at all
Package structure and language version	Painful to change once objects exist
Naming conventions	Renaming across hundreds of objects does not happen
Integration layer and pattern	Point-to-point interfaces accrete and are never retired
Extension governance and the gate	Retro-fitting a gate onto existing debt is a project
Master data governance approach	Data debt is harder to remediate than code debt

10.3 Phase-by-phase

Discover / Prepare

- Confirm the deployment model in writing (CC-A3)
- Agree the rule set and the exception process (CC-G1)
- Create the package structure with cloud language version (CC-D1)
- Configure ATC variants and activate the transport gate (CC-D4)
- Train the team on release contracts before they write anything

Gate: a developer cannot write non-compliant code without a syntax error.

Explore — the phase that decides the outcome

Fit-to-standard workshops. The discipline that matters:

- Every gap gets a **disposition** before it becomes a ticket
- The default disposition is **adopt standard**, and the burden of proof is on deviation, not on the standard
- A gap is only a gap after someone senior has asked “what happens if we just do it the standard way?”
- Record the rejected options for every accepted extension (CC-A4)

Gate: no development ticket exists without an approved gap disposition.

Realize

- Build to the rules
- Review against the checklist (Chapter 33.3)
- Performance design questions answered at design time (Chapter 26.5)
- Extension count tracked per process area, weekly

Gate: ATC clean, tests exist, volume tested.

Deploy

- Wrapper register complete, if any wrappers exist
- Exception register reviewed and each entry has a review date
- Gate ownership transferred to a named internal role (CC-G5)

Run

- See Chapter 12.7.

10.4 The gap disposition framework

Every gap from a fit-to-standard workshop gets exactly one of these. Force the choice in the workshop, not afterwards.

Disposition	Meaning	Target share
ADOPT	Change the process; use standard as delivered	60–75%
CONFIGURE	Standard covers it with configuration	10–20%
KEY USER	Field or simple logic via key user tools	5–10%
DEVELOP	On-stack developer extensibility required	5–10%
SIDE-BY-SIDE	Belongs on BTP	1–5%
DEFER	Real, but not for go-live	—
REJECT	Not justified by business value	—

Those percentages are targets to steer by, not measurements. **If DEVELOP exceeds 15%, stop and audit the workshops** — you are almost certainly recording process preferences as requirements. That single check, applied in week six rather than month six, is the most valuable intervention available on a greenfield project.

10.5 The gap register

The central artefact, and the greenfield equivalent of the disposition register in a conversion.

Field	Why
Gap ID, process area	Identity
Description as stated by the business	The raw requirement
Why standard does not cover it	Forces the question to be answered
Disposition	The decision
Options considered and rejected	CC-A4 evidence
Business owner	Accountability for the cost
Extension tier if DEVELOP	Compliance tracking
Approver and date	Governance

The “why standard does not cover it” field does most of the work. Requiring a written answer eliminates a meaningful share of gaps without any argument.

10.6 Greenfield metrics

- Gaps by disposition, tracked weekly through Explore
- Custom objects per process area — compare across areas; an outlier is a workshop problem, not a complexity problem
- Extensions by tier — tier 1 should be near 100% in greenfield
- Wrapper count — in a Public Cloud greenfield this must be zero; in Private Cloud, near zero, and every one is a question
- Standard Fiori apps adopted versus custom apps built

10.7 The conversation to have in week one

With the programme sponsor, before workshops start:

> *"Every gap we accept becomes a permanent cost. We will report gaps by disposition weekly. If development exceeds fifteen percent of gaps, that is a signal we are rebuilding the old system, and I will escalate it. Do you want that escalation to come to you?"*

Getting agreement to that sentence before the pressure starts is worth more than any technical control in this book.

Chapter 11: System Conversion (Brownfield)

Part VI is the remediation runbook. This chapter is about how remediation coexists with a conversion that has a hard cutover date, which is a different problem and the one that actually sinks programmes.

11.1 The two-clock problem

A conversion has a date. Clean core does not. Coupling them is the characteristic failure of brownfield programmes: the conversion slips because someone decided the system would be clean when it arrived.

Separate the backlogs. Completely.

	Conversion backlog	Clean core backlog
Driven by	S/4HANA readiness findings	ABAP Cloud readiness findings
Question	Will it work after conversion?	Is it upgrade-stable?
Deadline	The cutover date	Continuous
Blocking	Yes	No
Scope	Mandatory only	Prioritised by value

Run the two analyses separately, report them separately, and fund them separately. A merged report is the artefact that causes the coupling.

11.2 The brownfield clean core myth

You do not get clean at conversion. **You get converted, then you get clean.**

Anyone promising a clean core at cutover on a large brownfield estate is either scoping a two-year conversion or planning to fail. The honest sequence is:

1. Convert with mandatory remediation only
2. Stop the bleeding — governance and the gate, active from day one
3. Remediate in waves, post-conversion, funded by the retire savings

Saying this early costs you nothing and protects you when the schedule tightens. Saying it late makes you the person who changed the story.

11.3 Pre-conversion — the highest-leverage window

Most of the clean core value in a brownfield programme is available *before* the conversion, and it is routinely missed:

- **Retire the dead estate now.** Every object retired pre-conversion is an object that never has to be converted, tested or remediated. This is free scope reduction and it accelerates the conversion itself.
- **Activate the gate now** (CC-D4). New debt written during a two-year conversion programme is debt you will remediate later at full price.
- **Create the package structure and language version policy now** (CC-D1), so every object built for the conversion is already compliant.
- **Freeze what you will not remediate**, with review dates.

A programme that does only these four things pre-conversion arrives at cutover with a materially smaller estate and no new debt. That is a defensible outcome even if zero legacy objects were remediated.

11.4 During conversion

Discipline: **mandatory remediation only.** Clean core work during the conversion window competes with the cutover and loses, or wins and takes the date with it.

The exception: objects you are touching anyway for conversion reasons. If a program must be changed for a simplification item, remediate it fully while it is open. Marginal cost is low; doing it twice is expensive.

11.5 Post-conversion

Now Part VI applies in full — waves by functional domain, retire first, redesigns last. The advantages you now have that you did not before:

- A converted system, so released API availability is known rather than guessed
- Usage data on the new system, which differs from ECC usage
- A working gate, so the estate is no longer growing
- Business attention, because the go-live pain is fresh

Start the first wave within three months of hypercare closing. Wait longer and the organisation moves on, the budget goes elsewhere, and the programme never restarts.

11.6 The characteristic failures

- Coupling clean core to the cutover date, so the conversion slips
- Skipping pre-conversion retirement, so a bloated estate gets converted
- No gate during conversion, so two years of new debt arrives with the system
- No post-conversion wave, so “clean core” ends as a slide deck

Chapter 12: Other Project Architectures

Six more project shapes, each with a different answer to “where is clean core actually decided.”

12.1 Selective data transition

Shape. New S/4HANA system, selected configuration and data brought across from the legacy system. Neither greenfield nor conversion.

Where clean core is decided: at the selection. Every custom object brought across is a deliberate choice, which makes this the *best* project type for clean core — the default is that nothing comes unless someone argues for it.

Rules that bind hardest: CC-A3, CC-A4, and the greenfield gap disposition framework applied to migration candidates rather than to workshop gaps.

Characteristic failure: the selection is made by the technical team on technical criteria, so objects come across because they were easy to move rather than because the business needs them. Put the business owner in the selection decision, exactly as in a retire wave.

12.2 Template and rollout

Shape. A global template built once, rolled out to countries or entities.

Where clean core is decided: in the template. Whatever the template permits, every rollout will do — and then some. A template with fifty extensions becomes fifteen rollouts with eighty each.

The rule that matters most: local deviation governance. Every rollout will produce “local legal requirement” and “local business practice” gaps. The first category is real and non-negotiable. The second is where templates die.

Practical control: classify every local gap as *statutory*, *material*, or *preference*. Statutory is approved. Preference is refused. Material goes to the template board, and if approved it goes **into the template**, not into a local build. That last part is what stops fifteen divergent copies.

Characteristic failure: local extensions built locally, so the template becomes fiction within two rollouts and every future upgrade is fifteen separate regression exercises.

12.3 Upgrade-only projects

Shape. No conversion, no new scope. Moving from one S/4HANA release to a newer one.

Where clean core is decided: nowhere, by default — which makes this the cheapest possible entry point to introduce governance.

The opportunity: an upgrade already requires regression testing and a change freeze. Adding the ATC gate and the package/language version policy costs almost nothing incrementally, and the client is already in a change-control mindset. Every upgrade project should leave the client with a gate they did not have before.

Also do: review the wrapper register against newly released APIs (CC-G3). Each release publishes more, and each one lets you delete a wrapper. An upgrade is the natural moment.

12.4 Carve-out and divestiture

Shape. Part of the business separates, taking a system or a new system with it.

Where clean core is decided: in the scoping of what the new entity inherits. The new entity is effectively greenfield with a legacy shopping list.

Rules that bind: treat every inherited custom object as a migration candidate requiring justification (as 12.1). The carve-out is a rare licence to refuse inherited debt, and it expires the moment the new system goes live.

Characteristic failure: wholesale copy of the parent's custom estate under time pressure, importing twenty years of debt into a system that had a genuine chance to start clean.

12.5 Two-tier ERP

Shape. Headquarters on Private Cloud or on-premise, subsidiaries on Public Cloud.

Where clean core is decided: in the extension portability policy. The subsidiaries have no choice — Public Cloud enforces it. Headquarters does, and if HQ builds non-portable extensions, nothing can be shared downward.

The rule that matters most: CC-D1 applied at HQ voluntarily, plus the portability design points in Chapter 8.10. Build HQ extensions as if they were Public Cloud, and they can be deployed to the subsidiaries. Build them classically and every subsidiary re-implements from scratch.

Characteristic failure: HQ treats itself as exempt because it is Private Cloud, and the two-tier model degrades into two unrelated landscapes with duplicated development.

12.6 BTP-led and side-by-side-only

Shape. The core is frozen — often a legacy ECC or a heavily customised S/4HANA nobody will touch. All new capability is built on BTP.

Where clean core is decided: at the integration boundary, because that is the only part of the core being touched.

Rules that bind: CC-A6 and the whole of Chapter 18. This architecture is almost entirely an integration problem, and it fails as an integration problem.

Honest assessment: this is a containment strategy, not an architecture. It is right when the core genuinely cannot be changed on the available timeline. It is wrong as a permanent answer, because the extension estate on BTP grows a dependency on a core nobody is maintaining. If a client proposes this as their clean core strategy, ask what the plan for the core is. If there isn't one, say so.

12.7 Run and application management

Shape. No project. Business as usual, change requests, incidents.

Where clean core is decided: in the change request process. This is where every clean estate eventually gets dirty again, and it is the least glamorous and most important control in this book.

Rules that bind: CC-G4 and CC-G5 above all. If nobody internally owns the gate and nobody reports the object count, the debt returns within two years and nobody notices until the next upgrade.

The controls that actually work in run:

- The transport gate stays on. Permanently. No temporary disabling for a go-live crunch, because temporary is never temporary.
- Every change request over a size threshold gets an extension approach decision (CC-A4), not just an estimate.
- Wrapper and exception registers reviewed quarterly, with the review minuted.
- Custom object count on the monthly service report. Rising count is a reportable trend, not a curiosity.

Characteristic failure: the consultants leave, the gate is disabled during one crisis, and it is never re-enabled. Phase 10 in Chapter 27 covers the handover.

12.8 The summary matrix

Project type	Where clean core is won or lost	The one rule that matters most
Greenfield	Fit-to-standard workshops	Gap disposition discipline (10.4)
System conversion	Pre-conversion retirement and the gate	Separate the two backlogs (11.1)
Selective transition	The selection decision	Business owner signs every inclusion
Template / rollout	The template, and local deviation governance	Approved deviations go into the template
Upgrade	Introducing a gate that did not exist	Use the change freeze you already have
Carve-out	Scoping the inheritance	Refuse debt while you still can
Two-tier	HQ extension portability	HQ builds as if Public Cloud
BTP-led	The integration boundary	Ask what the plan for the core is
Run / AMS	The change request process	The gate stays on, owned by a named person

12.9 The question that identifies the project type

Clients rarely describe their project shape accurately. Ask instead:

> *"On the day you go live, will your custom objects be the ones you have today, a subset of them, or none of them?"*

The answer tells you which chapter applies, regardless of what the programme is called.

Part IV — Building Clean

Chapter 13: Package and Component Architecture

Structure decides everything downstream. Get it right on day one of a programme.

13.1 Software components

- `ZLOCAL` — the standard local software component for cloud development. Not transportable to another system as a component; used for on-stack extensions.
- **Customer software components** — created for BTP ABAP Environment or where you need independent lifecycle. Managed via the relevant Fiori app.
- `HOME` — the classic local component. Standard ABAP. Avoid for new work.

13.2 A package structure that survives

```
ZCC                          Root, structure package
├── ZCC_CORE                 Shared types, constants, utilities (tier 1)
├── ZCC_<DOMAIN>             One per business domain, e.g. ZCC_SD, ZCC_EWM
│   ├── ZCC_<DOMAIN>_MODEL   CDS views, data model
│   ├── ZCC_<DOMAIN>_BO      RAP behaviour, business objects
│   ├── ZCC_<DOMAIN>_UI      Service definitions, bindings, projections
│   └── ZCC_<DOMAIN>_TEST    Test doubles, test data
├── ZCC_EXT                  Extensions to SAP objects (CDS extends, BAdIs)
└── ZCC_WRAP                 Tier 2 wrappers — Standard ABAP, quarantined
```

Rules:

- `ZCC_WRAP` is the **only** Standard ABAP package. Everything else is ABAP for Cloud Development. This makes non-compliance visible in the package tree.
- Use package interfaces and use-access to enforce the dependency direction. Nothing outside `ZCC_WRAP` may be a Standard ABAP package.
- Domain packages must not depend on each other directly. Route through `ZCC_CORE` or released interfaces.

13.3 Naming conventions

Consistency matters more than the specific scheme. Agree one and enforce it in ATC.

Object	Pattern	Example
CDS interface view	`ZI_<Entity>`	`ZI_SalesOrderCredit`
CDS consumption view	`ZC_<Entity>`	`ZC_SalesOrderCredit`
CDS extension	`ZE_<SAPEntity>`	`ZE_I_SalesDocument`
Behaviour definition	matches root entity	
Service definition	`ZSD_<Scenario>`	
Service binding	`ZSB_<Scenario>_O4`	
Tier 2 wrapper	`ZCC_WRAP_<Capability>`	`ZCC_WRAP_PRICING`
Extension include	`ZZ<Field>`	`ZZCreditTier`

Customer field names in SAP structures conventionally start `ZZ` to avoid collision with SAP's own `Z` fields in some historical structures.

Chapter 14: CDS Extension Patterns

14.1 Extending an SAP CDS view

The mechanism is `EXTEND VIEW ENTITY` (for view entities) or `EXTEND VIEW` (for older DDIC-based CDS views). Confirm the SAP object carries **Extend in Cloud Development** before you start.

```
@EndUserText.label: 'Credit tier extension'
extend view entity I_SalesDocument with ZE_I_SalesDocument
{
  _SoldToParty.ZZCreditTier as ZZCreditTier
}
```

Rules:

- One extension object per SAP view per purpose. Do not accumulate unrelated fields in one extension.
- Never add an aggregation or a join that changes the cardinality of the base view. You will silently break every consumer.
- Watch performance: an extension that adds an association traversal executes for every consumer of the base view, including SAP's own apps.
- Extension fields must be added to the underlying table first (append structure or extension include) before they can be surfaced.

14.2 The layered CDS model

Follow the virtual data model conventions even in custom code:

- **Basic / interface views (`ZI_`)** — reusable, no UI annotations, stable field names, one per business entity
- **Composite views** — joins and aggregations over interface views

- **Consumption views (`zc_`)** — UI annotations, one per app, disposable

The discipline that pays: **UI annotations never appear in interface views.** When the UI changes you touch a consumption view only.

14.3 Metadata extensions

Put UI annotations in a metadata extension rather than inline, so the annotations can be changed and transported independently, and so a client adaptation does not require editing your view.

```
@Metadata.layer: #CUSTOMER
annotate entity ZC_SalesOrderCredit with
{
  @UI.lineItem: [{ position: 10 }]
  SalesOrder;
}
```

Chapter 15: RAP for Extension Scenarios

You do not need to be a RAP framework expert to be a clean core specialist, but you need the extension-relevant parts cold.

15.1 The two flavours

- **Managed** — RAP handles persistence. Use for new custom business objects.
- **Unmanaged** — you implement the interaction and save phases yourself. Use when wrapping existing logic or legacy persistence.

For clean core work, the third case matters most:

- **Managed with unmanaged save** — RAP handles the interaction phase, you implement save. This is the common pattern when the actual persistence is an existing API or a legacy update mechanism.

15.2 Behaviour extensions

To add behaviour to an SAP RAP business object, use `extend behavior definition`. The SAP BO must be released for extension.

```
extend behavior for I_SalesDocumentTP
{
  field ( readonly ) ZZCreditTier;
  validation validateCreditTier on save { field ZZCreditTier; }
}
```

Behaviour extensions are the cloud-compliant replacement for a large class of classic BAdI implementations. When remediating, check whether the classic BAdI you are replacing has a RAP behaviour extension equivalent — often it does.

15.3 The projection layer

Never expose an interface view directly as a service. Always project:

```
ZI_Entity  →  ZC_Entity (projection)  →  ZSD_Service  →  ZSB_Service
```

This gives you a place to change field exposure, add UI semantics and version the service without touching the model.

15.4 Business events

RAP business events are the cloud-compliant way to notify other systems. Prefer them over:

- Custom change pointers
- Update function modules that call an interface
- Database triggers (which do not exist in this world anyway)

Events decouple. In a clean core architecture, an event published from the core and consumed on BTP is the canonical integration pattern.

Chapter 16: Key User Extensibility

Do not skip this. Every hour of developer time spent on something a key user tool handles is waste, and knowing the boundary is part of the expertise.

16.1 What key user tools cover

- **Custom Fields and Logic** — add fields to standard business contexts, publish them to UIs, reports, email templates and APIs; implement logic in key user BAdIs
- **Custom CDS Views** — build analytical and reporting views over released entities
- **Custom Business Objects** — lightweight custom entities with generated UI and OData
- **Custom Analytical Queries** — query layer over cubes
- **Adaptation / UI Flexibility** — change Fiori app layout without code
- **Custom Communication Scenarios** — expose custom services for integration
- **Custom Reusable Elements** — shared logic across custom fields

16.2 The boundary

Push to key user when: the change is field-level, the logic is simple, the business wants to own it, and no complex data access is needed.

Escalate to developer extensibility when: you need reuse across many contexts, complex ABAP, custom persistence, performance-critical code, or proper unit testing. The key user editor has no test framework worth the name — this alone disqualifies it for anything with real business risk.

16.3 The governance failure

Key user extensibility is ungoverned by default. Business users create fields and logic that nobody inventories, nobody tests and nobody documents. On a mature system this becomes a shadow custom codebase.

Include key user extensions in your inventory scope. Most assessments miss this entirely, which is an easy differentiator on a client pitch.

Chapter 17: Security and Data Protection Architecture

The area architects are held accountable for and technical consultants most often skip. Everything here is auditable, and an auditor's question is not "does it work" but "show me the design."

17.1 Identity types

Get this right first; almost every downstream security defect traces back to the wrong identity type.

Type	Used for	Authenticates as
Business / dialog user	A human using a Fiori app	Named person, SSO or password
Communication user	System-to-system inbound calls	Certificate or basic auth, scoped to a communication scenario
Technical / background user	Scheduled jobs, internal processing	No interactive logon

Rules: never let a system-to-system integration authenticate as a named person. Never let a dialog user hold background authorisations "because it was easier." One communication user per scenario, not one shared across all of them — otherwise you cannot revoke or audit anything.

17.2 Authorisation design

Concern	PCE	Public
Role construction	PFCG roles, authorisation objects	Business roles built from business catalogues
Custom authorisation object	Available	Available with restrictions
Restriction on data	Authorisation object + explicit check	Restriction types on business roles
Who maintains	Security team, transported	Key user, configured

17.3 CDS access control (DCL)

Authorisation on a CDS view is declarative and is the primary mechanism in a cloud world. Set the annotation deliberately:

```
@AccessControl.authorizationCheck: #CHECK
define view entity ZI_CreditTolerance as select from zcredit_tol { ... }
```

- `#CHECK` — a DCL role must exist and is enforced. The default for anything reading business data.
- `#NOT_REQUIRED` — no DCL needed; the consumer is responsible. Use only for technical or fully public views, and justify it.
- `#NOT_ALLOWED` — no check performed. Requires a very good reason.

The trap: DCL is *not* automatically inherited up a view stack in the way people assume. A view built over a protected view can expose data the protection was meant to prevent. Verify the effective authorisation at the consumption layer, not the basic layer, and test it with a restricted user.

17.4 RAP authorisation

Two levels, and they answer different questions:

- **Global authorisation** — may this user perform this operation at all? Evaluated once per request.
- **Instance authorisation** — may this user perform this operation on *this instance*? Evaluated per instance.

```
define behavior for ZI_CreditTolerance alias Tolerance
authorization master ( instance )
{
  update;
  delete;
  field ( readonly ) CustomerId;
}
```

Instance authorisation costs per row. Do not implement instance checks that could be expressed globally — this is a common and expensive design error that shows up as list-view slowness.

17.5 Secrets and credentials

- **Never hardcode credentials.** In cloud ABAP the mechanisms exist precisely so you do not have to.
- Inbound and outbound credentials live in communication arrangements, not in custom tables.
- BTP-side secrets live in the destination service or the credential store.
- Certificate-based authentication over basic auth wherever the counterparty supports it.
- Rotation must be a documented procedure with an owner, not an incident.

17.6 Principal propagation to BTP

When a side-by-side extension acts on behalf of a user, the user's identity must travel with the call. Design decisions to make explicitly:

- OAuth SAML bearer flow versus a technical user, per integration
- Identity provider and trust configuration between the ABAP system and the BTP subaccount
- Cloud Connector configuration for on-premise reachability
- What the extension is authorised to do when the user identity is absent — usually the answer should be "nothing"

The failure mode: an extension that runs as a powerful technical user because principal propagation was "too complicated to configure." That converts every BTP application vulnerability into a full ERP compromise.

17.7 Data protection and privacy

Concern	Mechanism
Identify personal data	`@PersonalData.*` annotations on CDS entities and fields
Who read personal data	Read Access Logging, configured for the sensitive fields
Who changed data	Change documents; RAP change document integration
Retention and deletion	ILM objects, blocking and deletion rules
Purpose limitation	Purpose-based access, where the client's regime requires it

Annotate personal data in **your own custom entities** as well as consuming SAP's. A custom table holding contact details with no privacy annotation is a finding in any competent audit, and it is the specific thing custom code is most often criticised for.

17.8 Static security analysis

ATC includes security and code vulnerability checks. Enable them in `Z_CLOUD_STRICT`. They catch:

- Injection risks in dynamic SQL and dynamic method calls
- Directory traversal in file handling
- Missing authority checks on sensitive operations
- Hardcoded credentials
- Unsafe use of dynamic programming

The cloud language version removes many of these categories by construction, which is a genuine security benefit worth citing when justifying ABAP Cloud to a security team. It does not remove all of them.

17.9 Security review checklist

- ☐ Identity type correct for every caller
- ☐ One communication user per scenario
- ☐ `@AccessControl.authorizationCheck` set deliberately on every view
- ☐ DCL effective at the consumption layer, tested with a restricted user
- ☐ RAP global versus instance authorisation chosen on cost as well as need
- ☐ No hardcoded credentials anywhere
- ☐ Principal propagation designed, not defaulted to a technical user
- ☐ Personal data annotated on custom entities
- ☐ Read access logging configured where required
- ☐ Retention and deletion rules defined for custom persistence
- ☐ ATC security and vulnerability checks clean

Chapter 18: Integration Architecture

Chapter 8 asserts that integration is usually a bigger Public Cloud blocker than custom code. This chapter is the substance behind that claim.

18.1 The first decision: synchronous or asynchronous

Choose synchronous when	Choose asynchronous when
The caller needs the result to continue	The caller does not need an immediate answer
The operation is a query	The operation is a notification of a fact
Failure should surface to the user immediately	Failure should be retried without user involvement
Volume is low and predictable	Volume is high or bursty
Coupling to the counterparty's availability is acceptable	It is not

Default to asynchronous for anything that is a notification. Synchronous integration couples your availability to theirs, and most "we need it immediately" requirements do not survive the question "what happens if the other system is down for four hours?"

18.2 Inbound

- Model the service: CDS → service definition → service binding (OData V4 for new work)
- Expose it through a **communication scenario**, then a communication arrangement per counterparty
- One scenario per logical interface, not one catch-all scenario
- Version the service explicitly; do not silently change an exposed contract

18.3 Outbound

- HTTP calls through released HTTP client APIs, never raw sockets
- Destination configuration external to the code — the endpoint is configuration, not a constant
- Timeouts set explicitly. An unset timeout is an outage waiting for a slow counterparty.
- Circuit-breaker behaviour for repeated failures, so one dead endpoint does not consume all work processes

18.4 Event-driven

RAP business events are the cloud-native mechanism.

- Publish events for **facts**, not for commands. “SalesOrderBlocked” not “PleaseBlockSalesOrder.”
- Topic naming: a stable, hierarchical convention agreed once. Renaming a topic later breaks every subscriber.
- Payload: enough for the subscriber to act, or a reference the subscriber can dereference. Choose deliberately — fat payloads couple schemas, thin payloads cause call-backs.
- Events are typically **at-least-once**. Subscribers must be idempotent. Design for duplicate delivery rather than hoping.

18.5 The reliability patterns

Every asynchronous integration needs an answer to each of these. “We’ll handle it later” is how integration estates become unmaintainable.

Concern	Design question
Idempotency	If this message arrives twice, what happens?
Ordering	Does order matter? If yes, how is it guaranteed?
Retry	How many times, with what backoff?
Dead letter	Where does a permanently failing message go, and who looks at it?
Replay	Can we reprocess after fixing a bug? From where?
Monitoring	Who is alerted, on what signal, within what time?
Backpressure	What happens when the consumer is slower than the producer?

18.6 Migrating a point-to-point estate

The realistic path for a client with two hundred RFC and file interfaces:

1. **Inventory.** Every interface, its counterparty, volume, criticality, transport mechanism and owner. This is usually the first time anyone has listed them.
2. **Classify.** Retire (nobody uses it), replace (standard API exists), re-platform (move to the integration layer as is), redesign (rebuild as API or event). The same disposition discipline as custom code.
3. **Introduce the integration layer** before migrating anything, so there is somewhere to migrate to.
4. **Strangler pattern.** Put the middleware in front of the existing interface, cut counterparties over one at a time, then remove the legacy endpoint. Never a big-bang cutover on integration.
5. **Sequence by risk**, lowest first, to build operational confidence in the new layer before moving anything critical.

This is a programme in its own right, often larger than the custom code remediation. Scoping it as a workstream of the ABAP programme is a common and expensive error.

18.7 Anti-patterns

- **Database-level integration.** Another system reading your tables directly. Impossible in Public Cloud, and it should be impossible by policy everywhere else.
- **Chatty interfaces.** One call per line item instead of one call per document. The single most common cause of integration performance problems.
- **Business logic in the middleware.** The mapping layer accretes rules that belong in one system or the other, and nobody can find them later.
- **Interfaces with no owner.** If nobody is named, nobody watches the error queue.
- **Synchronous chains.** A calls B calls C calls D. Availability multiplies downward and the whole chain is as reliable as its worst link.

18.8 Integration review checklist

- ☐ Synchronous versus asynchronous chosen deliberately, with a reason
- ☐ Contract versioned, with a deprecation policy
- ☐ One communication scenario per logical interface
- ☐ Timeouts and retry policy explicit
- ☐ Idempotency designed on every asynchronous consumer
- ☐ Dead letter destination defined, with a named owner
- ☐ Replay path exists and has been tested
- ☐ Monitoring and alerting configured
- ☐ Payload granularity avoids chattiness
- ☐ No business logic in the mapping layer
- ☐ No direct database access from outside

Chapter 19: Testing Architecture

The manual states repeatedly that testing is 40–60% of remediation effort. This chapter is how it is actually done, and it specifies the semantic equivalence harness the rest of the book keeps referring to.

19.1 The pyramid

Level	What	Volume	Speed
Unit	ABAP Unit with test doubles, no database	Hundreds to thousands	Seconds
Component	CDS and RAP with test doubles for dependencies	Dozens to hundreds	Seconds to minutes
Integration	Real data, real services, cross-object	Dozens	Minutes
End-to-end / UI	Full business process through the app	A few	Slow, expensive

Most SAP projects invert this: a handful of manual end-to-end tests and nothing below. That inversion is precisely why remediation is slow and why regressions reach

production. Fixing it early is the highest-leverage thing a programme can do, and it is the argument you should make in week one.

19.2 ABAP Unit

```
CLASS ltc_credit_check DEFINITION FINAL FOR TESTING
  DURATION SHORT RISK LEVEL HARMLESS.
  PRIVATE SECTION.
    DATA mo_cut TYPE REF TO zcl_credit_check.
    METHODS setup.
    METHODS exposure_above_tolerance FOR TESTING.
ENDCLASS.
```

- `RISK LEVEL HARMLESS` means no persistent change. Anything else must be justified, because it constrains where the test can run.
- `DURATION SHORT` keeps it in the fast suite. Slow tests get excluded from the gate and then stop being run at all.
- One assertion concept per test. A test that checks five things tells you nothing useful when it fails.

19.3 Test doubles

The frameworks that make ABAP testing viable without a populated system:

Framework	Doubles	Use for
ABAP test double	Interfaces and classes	Isolating the unit under test from collaborators
CDS test double	CDS entities	Testing a view's logic with controlled input data
Open SQL test double	Database tables	Testing code that reads tables directly
RAP BO test double	Business objects	Testing EML consumers without touching real BOs

The design consequence: code that is hard to double is hard to test. Depend on interfaces, inject collaborators, and keep static access out of business logic. If a class cannot be constructed without a live system, that is a design defect, not a testing problem.

19.4 Test data management

- **Subset, do not copy.** A full production copy is expensive, slow and a data protection exposure.
- **Anonymise**, with the client's approved method, and document it.
- **Cover the shapes that break things**: reversals, cancellations, multiple currencies, multiple company codes, period boundaries, missing master data, maximum-length fields.

- **Version the test data** alongside the code. A test that passes only against a snapshot nobody can reproduce is not a test.
- Define a **refresh cadence** and who owns it.

19.5 The semantic equivalence harness

Referenced throughout Part V. Here is what it actually is.

Purpose. Prove that remediated code produces the same business result as the code it replaces, on realistic data, before it is transported.

Design.

1. **Dual execution.** Run the legacy implementation and the new implementation against the same input set, in the same system state.
2. **Canonicalisation.** Normalise both result sets before comparing: sort order, trailing spaces, null versus initial, floating-point rounding, timestamp fields. Without this step, everything looks different and the harness gets abandoned in week two.
3. **Diff.** Compare row by row, field by field. Report differences with enough context to diagnose.
4. **Tolerance rules.** Agreed in advance, per field: which differences are acceptable and why. Rounding to two decimals, yes. A missing document, no.
5. **Runtime capture.** Record execution time for both. A functionally equivalent replacement that is four times slower is still a defect (Chapter 26.7).
6. **Report.** Object, input set size, differences by category, runtime delta, verdict.

Build it once, at the start of the programme. Every remediated object runs through it. A team that builds this in phase 5 remediates at several times the speed of a team that tests each object by hand, and produces a defensible audit trail as a by-product.

19.6 Regression suite design

- Group by **business process**, not by object type — so a wave can be signed off by one business area.
- Every defect found in test or production becomes a test case. This is the only mechanism that stops the same defect recurring.
- Keep the fast suite genuinely fast, or it will be skipped under deadline pressure. That is when you most need it.

19.7 Testing in the pipeline

```
commit → syntax → ATC (Z_CLOUD_STRICT) → ABAP Unit (fast suite)
       → transport → integration suite → equivalence harness → sign-off
```

Coverage targets: aim for meaningful coverage of business logic rather than a headline percentage. 80% coverage of getters and setters is worthless; 40% coverage that hits every branch of the pricing rule is valuable. Report what is covered, not just how much.

19.8 Testing review checklist

- ☐ Unit tests exist and run without a populated system
- ☐ Test doubles used rather than live dependencies
- ☐ `RISK LEVEL` and `DURATION` set appropriately
- ☐ Edge-case data shapes covered
- ☐ Equivalence harness run for any remediated object
- ☐ Tolerance rules agreed and documented
- ☐ Runtime compared, not only output
- ☐ Every consumer from the where-used graph regression tested
- ☐ New defects converted into test cases

Chapter 20: Worked Example — End to End

Everything before this is principles. This chapter runs one requirement through all of them. The value is in the decision points, not the code.

20.1 The requirement

> *Block creation of a sales order when the customer's overdue exposure exceeds a negotiated tolerance held per customer. The credit team needs an app to maintain tolerances and review blocked orders. Notify the collections system when an order is blocked.*

Non-functional facts gathered before designing — ask for these every time:

Question	Answer
Sales orders per day	~8,000, peak 20,000 at month end
Customers with a tolerance	~4,000
Acceptable added latency on order save	< 200 ms
Tolerance maintenance users	12
Deployment model	Private Cloud, Public Cloud not ruled out
Data protection	Customer credit data is restricted to the credit team

The latency budget and the "Public Cloud not ruled out" answer shape almost every decision that follows.

20.2 Prerequisite check (Chapter 5)

Target release confirmed. Package structure exists. Cloud language version policy in force. ATC gate active. Equivalence harness not needed — this is new development, not remediation.

20.3 Extensibility decision (Chapter 7)

1. **Standard configuration?** Standard credit management covers exposure, but not this client's negotiated per-customer tolerance concept. Documented as assessed and rejected — always record this, it is the first question an architect will ask.
2. **Key user?** Tolerance could be a custom field. But the blocking logic needs to read across open items and aggregate, which exceeds custom logic's practical scope, and there is no test framework. Rejected.
3. **Developer extensibility?** Yes. Needs core data with no latency and transactional consistency with the order save.
4. **Side-by-side?** No. A 200 ms budget on save rules out a network round trip to BTP. **This is the decision most likely to be got wrong**, because side-by-side is the fashionable answer.

Verdict: on-stack developer extensibility.

20.4 Deployment model impact (Chapter 8)

Aspect	PCE (chosen)	Public Cloud, if it moved
Tolerance persistence	Custom table, C1-released	Same
Blocking hook	Released BAdI or behaviour extension preferred; classic BAdI acceptable as tier 2	Released hook only — if none exists, redesign
Notification	RAP business event	Same
App	Fiori elements over RAP	Same

Portability decision: build as if Public Cloud, so the only PCE-specific artefact is the hook, if a tier 2 hook proves necessary.

20.5 Package structure (Chapter 13)

```
ZCC_CREDIT                  (ABAP for Cloud Development)
├── ZCC_CREDIT_MODEL        CDS views, table
├── ZCC_CREDIT_BO           Behaviour, validation logic
├── ZCC_CREDIT_UI           Projection, service definition, binding
└── ZCC_CREDIT_TEST         Test doubles, test data
ZCC_WRAP                    (Standard ABAP — only if the hook forces it)
```

20.6 Data model (Chapters 10, 19)

Custom persistence for the tolerance:

```
@AccessControl.authorizationCheck: #CHECK
@PersonalData.entitySemantics: #DATA_SUBJECT_DETAILS
define root view entity ZI_CreditTolerance
  as select from zcc_credit_tol
  association [0..1] to I_Customer as _Customer
    on $projection.CustomerId = _Customer.Customer
{
  key customer_id      as CustomerId,
      tolerance_amount as ToleranceAmount,
      currency         as Currency,
      valid_from       as ValidFrom,
      _Customer
}
```

Exposure, computed on the database:

```
@AccessControl.authorizationCheck: #CHECK
define view entity ZI_CustomerOverdueExposure
  as select from I_OperationalAcctgDocItem
{
  key Customer        as CustomerId,
      currency        as Currency,
      sum( AmountInCompanyCodeCurrency ) as OverdueAmount
}
where NetDueDate < $session.system_date
  and ClearingDate is initial
group by Customer, currency
```

Design points worth defending in review:

- Aggregation on the database, not by reading open items into ABAP (Chapter 21.3)
- `_Customer` is an **association, not a join** — consumers that do not need customer attributes pay nothing (Chapter 23.4)
- `@AccessControl.authorizationCheck: #CHECK` on both, with DCL restricting to the credit team (Chapter 17.3)
- `@PersonalData` annotation on the entity holding customer credit terms (Chapter 17.7)

20.7 The maintenance app (Chapter 15)

Managed RAP BO on `ZI_CreditTolerance`, projection `ZC_CreditTolerance`, service definition, OData V4 binding, Fiori elements list report / object page.

```
managed implementation in class zbp_i_credittolerance unique;
strict ( 2 );

define behavior for ZI_CreditTolerance alias Tolerance
persistent table zcc_credit_tol
lock master
authorization master ( global )
{
  create; update; delete;
  field ( readonly ) CustomerId;
  validation validateTolerance on save { field ToleranceAmount; }
}
```

Decision point: `authorization master ( global )` rather than instance — the credit team either maintains tolerances or does not. Instance-level checks would cost per row on a

4,000-row list for no functional benefit (Chapter 17.4). No draft: twelve users editing single records occasionally do not need it, and draft doubles persistence (Chapter 25.5).

20.8 The blocking logic

Order of preference, worked through explicitly:

1. **Released RAP behaviour extension on the sales order.** If the sales order BO is released for extension, add a validation. Fully tier 1. **Check this first — people assume it is unavailable and it increasingly is not.**
2. **Released BAdI.** Tier 1.
3. **Classic BAdI, delegating immediately.** Tier 2, and only the hook:

```
METHOD if_ex_sales_order_check~check.
  " Tier 2 hook — no logic here, by design
  DATA(lo_check) = NEW zcl_cc_credit_block_check( ).
  rv_blocked = lo_check->is_blocked( iv_customer = is_header-kunnr
                                     iv_amount   = is_header-netwr ).
ENDMETHOD.
```

All logic sits in `zcl_cc_credit_block_check`, a tier 1 class in `ZCC_CREDIT_BO`. If SAP later releases a proper hook, two lines change and the logic is untouched. The wrapper goes in the register (Appendix C) with replacement condition: *"remove when a released validation extension exists on the sales order BO."*

20.9 Performance design (Part IV)

Question	Answer
Volume	Runs on every order save — 20,000 times on peak day
Access pattern	Single customer, single tolerance read, single exposure read
Table type	Tolerance cache: hashed by customer (Chapter 24.1)
Round trips	Two reads per save; both single-row by key
Risk	The exposure aggregate over open items

The exposure view aggregates open items per customer. At 20,000 calls a day that is 20,000 aggregations. **Measured before deciding**: 40 ms per call against production-like volume, within the 200 ms budget. Recorded, no optimisation applied.

Had it exceeded budget, the options in order would have been: restrict the aggregate further, materialise exposure periodically and accept staleness, or move the check to a later point in the process. Not "add an index."

20.10 Integration (Chapter 18)

Blocking publishes a RAP business event, consumed by collections through Event Mesh.

- A **fact**, not a command: `SalesOrderCreditBlocked`
- Payload: order ID, customer, exposure, tolerance, timestamp
- Asynchronous — collections does not need to answer for the order to be blocked, and the save must not depend on their availability
- Consumer must be idempotent; at-least-once delivery assumed

20.11 Testing (Chapter 19)

- Unit tests on `zcl_cc_credit_block_check` with a CDS test double supplying controlled exposure and tolerance rows. No live data.
- Cases: no tolerance defined, exposure below, exposure exactly equal, exposure above, currency mismatch, customer not found, zero tolerance.
- RAP BO tests on the maintenance object.
- Integration test for the event publication.
- Not required: the equivalence harness. New development, nothing to compare against.

20.12 Delivery

ATC clean against `Z_CLOUD_STRICT` including security checks. Unit tests in the fast suite. Transport gated. Wrapper registered if used. ADR written (Appendix F).

20.13 The four decisions that mattered

Strip away the code and this is what an architect is actually judged on:

1. **Rejecting side-by-side on a latency budget.** The fashionable answer was wrong, and the reason was a number gathered before designing.
2. **Association rather than join**, so consumers that do not need customer data pay nothing.
3. **Global rather than instance authorisation**, a security decision made on cost as well as need.
4. **Thin hook, tier 1 logic**, so the only non-compliant artefact is two lines with a documented removal condition.

Each is defensible in one sentence with a reason. That is the standard.

Part V — Performance Engineering

Compliance and performance are independent. Code can pass every ATC cloud check and still bring a system down. This part is the other half of “good.”

Chapter 21: The Performance Model

21.1 What actually changed with HANA

Most performance advice in circulation is pre-HANA and now actively wrong. The differences that matter:

Classic DB assumption	HANA reality
Joins are expensive; denormalise	Joins are cheap; aggregates and redundant tables were removed for this reason
Aggregation is expensive; pre-aggregate	Aggregation is cheap and parallel; do it on the database
Add secondary indexes freely	Rarely needed; columnar storage scans fast. Indexes cost memory and write time
Reading few columns from many rows is expensive	This is the *optimal* access pattern for columnar storage
Reading all columns of one row is cheap	Relatively more expensive — column store must reassemble the row
Sorting in ABAP may beat sorting on DB	DB sort is almost always faster

The consequence: the correct pattern is now *narrow and set-based* — few columns, many rows, aggregated and filtered on the database. `SELECT *` and row-by-row processing are worse on HANA than they were on a classic database, not better.

21.2 The three costs

Every operation costs in one or more of three currencies. Diagnose by asking which one is dominant before you change anything.

1. **Database time** — how long the DB spends producing the result.
2. **Transfer volume** — how much data crosses from DB to application server. This is the cost most often ignored and most often dominant.
3. **Application server CPU and memory** — ABAP-side processing, internal table operations, and memory consumption.

A statement that is fast on the database but returns two million rows to ABAP is a performance defect even though the SQL trace looks clean.

21.3 Code-to-data

The single organising principle: **move the calculation to the data, not the data to the calculation.**

In practice this means, in descending order of preference:

1. Do it in a CDS view — filters, joins, aggregations, calculations
2. Do it in Open SQL — aggregate functions, `GROUP BY`, expressions
3. Do it in AMDP — only for genuinely set-based algorithms that CDS cannot express, and only over C4-released artefacts
4. Do it in ABAP — last resort, and only for logic that is genuinely procedural

The anti-pattern is reading raw rows into an internal table and computing in ABAP what the database could have computed. It is the most common performance defect in custom code and it is exactly what most legacy ABAP does.

21.4 The cloud-specific constraint

ABAP for Cloud Development removes some traditional escape hatches. You cannot use native SQL, you cannot bypass released views to read the raw table, and in Public Cloud you cannot create your own secondary indexes on SAP tables. This means **the design of your data access has to be right, because you cannot tune your way out of it afterwards.**

That is a real constraint and worth stating to an architect early: in a cloud world, performance is a design-time concern far more than a tuning-time one.

Chapter 22: Every Area of Checking

The full checklist. Work it in design review and in code review. Each area lists what to check and the specific defect to look for.

22.1 Area A — Database access

- ☐ No `SELECT` inside a `LOOP`. Ever. This is defect number one.
- ☐ No `SELECT *` — name the fields you need
- ☐ `WHERE` clause present and selective; no filtering after the fact in ABAP
- ☐ Aggregation done on the database, not by looping and summing
- ☐ `SELECT SINGLE` used only when the key is fully qualified
- ☐ `FOR ALL ENTRIES` driver table checked for emptiness first
- ☐ `FOR ALL ENTRIES` duplicates understood — the statement removes them
- ☐

No `ORDER BY` omitted where the result order is relied upon

- ☐ `UP TO n ROWS` combined with `ORDER BY` where "top n" is meant
- ☐ No nested `SELECT` where a join or subquery would do
- ☐ Number of round trips minimised — one set operation beats a hundred singles
- ☐ `INTO TABLE` rather than `INTO` inside a loop
- ☐ Package-size processing used for genuinely large result sets

22.2 Area B — CDS view design

- ☐ View stack depth reasonable — not a view on a view on a view on a view
- ☐ Associations used instead of joins where the field may not be needed
- ☐ No join that changes cardinality unintentionally
- ☐ Cardinality declared on associations so the optimizer can eliminate them
- ☐ No `UNION` where a filter or a `CASE` would do
- ☐ No calculated expression inside a join condition
- ☐ Currency and quantity conversion not applied in high-volume interface views
- ☐ Client handling correct and not duplicated
- ☐ Extensions reviewed for cost — an extension executes for *every* consumer
- ☐ View entities preferred over DDIC-based CDS views for new development
- ☐ Filters pushed to the lowest view in the stack that can hold them
- ☐ Analytical views annotated correctly so the analytic engine is used

22.3 Area C — Internal table processing

- ☐ Table type matches the access pattern (see 20.1)
- ☐ No `READ TABLE ... WITH KEY` on a large standard table
- ☐ No nested loop over two large tables — use a hashed lookup or parallel cursor
- ☐ `ASSIGNING` or `REFERENCE INTO` instead of `INTO` where the row is large
- ☐

`DELETE ADJACENT DUPLICATES` preceded by the matching `SORT`

- ☐ Secondary keys defined where a second access path is genuinely needed
- ☐ No `COLLECT` on large volumes — aggregate on the database instead
- ☐ Tables cleared or freed when no longer needed in long-running code

22.4 Area D — Memory

- ☐ No unbounded internal table — always a filter or a package size
- ☐ Large tables `FREE`d rather than `CLEAR`ed when finished
- ☐ Deep structures avoided where flat would do
- ☐ String handling not creating repeated copies in loops
- ☐ No accumulation across a loop that could be streamed

22.5 Area E — Control flow

- ☐ Loop nesting depth justified
- ☐ Expensive operations hoisted out of loops
- ☐ Early exit (`EXIT`, `CHECK`, `RETURN`) used where possible
- ☐ `CASE` preferred to long `IF`/`ELSEIF` chains on a single value

22.6 Area F — Buffering and reuse

- ☐ Repeated identical reads within one request cached in memory
- ☐ Configuration and text reads not repeated per line item
- ☐ Table buffering considered for small, read-mostly custom tables
- ☐ No buffering on tables that are written frequently

22.7 Area G — RAP, OData and UI

- ☐ Determinations scoped to `on modify` where they need not run on save
- ☐ Determinations and validations do not re-read the whole business object
- ☐ Value help views lean and filtered
-

- ☐ `$expand` depth controlled — each level multiplies cost
- ☐ Server-side paging in place; no unbounded list retrieval
- ☐ Draft handling used only where genuinely required — it doubles persistence
- ☐ Field control evaluated at the right granularity, not per instance where static would do
- ☐ Analytical requirements served by an analytical query, not a transactional service

22.8 Area H — Locking and LUW

- ☐ Locks held for the shortest possible time
- ☐ No user interaction while a lock is held
- ☐ Lock granularity appropriate — not locking a whole table
- ☐ `COMMIT WORK` not issued inside a loop over items
- ☐ Update task semantics preserved when wrapping legacy update logic

22.9 Area I — Background and mass processing

- ☐ Mass jobs process in packages, not in one transaction
- ☐ Restartability designed in — a failed job can resume
- ☐ Parallelisation considered for genuinely independent units
- ☐ Job runtime measured against the business window it must fit

22.10 Area J — Authorisation and logging

- ☐ Authority checks outside loops where the check is invariant
- ☐ Application logging not written per item in high-volume processing
- ☐ Trace and debug statements removed before transport

Chapter 23: SQL and CDS Performance in Depth

23.1 The five golden rules

SAP's classic formulation, still correct:

1. **Keep the result set small.** Filter on the database.
2. **Keep the transferred data small.** Select only needed columns; aggregate before transferring.
3. **Reduce the number of database executions.** Set operations, not loops.
4. **Reduce the search effort.** Filter on fields the database can restrict on efficiently.
5. **Keep unnecessary load away from the database.** Do not re-read what you already have.

Rules 2 and 3 are where most defects live. Rule 4 matters less on HANA than it did, but selectivity still matters on very large tables.

23.2 `SELECT` in a loop — the fix hierarchy

The defect:

```
LOOP AT lt_items INTO DATA(ls_item).
  SELECT SINGLE * FROM zmaterial
    INTO @DATA(ls_mat)
    WHERE matnr = @ls_item-matnr.
  ...
ENDLOOP.
```

Fix options, in order of preference:

1. **Join or CDS view.** Model the relationship once and read it in a single statement. Best answer in almost every case.
2. **Single set read into a hashed table**, then look up inside the loop:

```
SELECT matnr, matkl, meins
  FROM zmaterial
  FOR ALL ENTRIES IN @lt_items
  WHERE matnr = @lt_items-matnr
  INTO TABLE @DATA(lt_mat).

DATA(lt_mat_hash) = ... " hashed by matnr

LOOP AT lt_items ASSIGNING FIELD-SYMBOL(<item>).
  READ TABLE lt_mat_hash WITH TABLE KEY matnr = <item>-matnr
       ASSIGNING FIELD-SYMBOL(<mat>).
  ...
ENDLOOP.
```

1. **CDS view with a parameter or an association path expression**, where the consumer needs the joined result anyway.

23.3 `FOR ALL ENTRIES` — the traps

Useful, and full of sharp edges:

- **Empty driver table returns everything.** Always check `IF lt_driver IS NOT INITIAL` first. This is the single most common production incident from otherwise reasonable code.
- **Duplicates are removed** from the result. If you need the multiplicity, you must re-join in ABAP.
- **`ORDER BY` is not reliable** — sort afterwards, or restructure.
- **The driver table should contain only the key fields**, deduplicated, before the statement.
- It is executed as multiple database calls under the covers; a join is usually better when both tables are available.

Prefer a join, a subquery, or a CDS association. `FOR ALL ENTRIES` is the answer when the driver set comes from somewhere the database cannot see.

23.4 CDS view stacking

The most common CDS performance defect is depth. A consumption view over a composite view over three interface views over four basic views produces a query the optimizer must flatten, and every unnecessary level adds fields, joins and conversions that nobody needs.

Rules that keep the stack fast:

- Push filters **down** to the lowest view that can hold them. A `WHERE` on the consumption view still has to be pushed by the optimizer; help it.
- Use **associations rather than joins** for optional relationships. An association is only materialised when a path expression uses it, so consumers that do not need the field pay nothing. This is the single most valuable CDS performance technique.
- Declare **cardinality** accurately. It lets the optimizer eliminate joins it can prove are unnecessary.
- Avoid `UNION`. It blocks several optimisations. A `CASE` or an additional filter is usually expressible instead.
- Never put a calculated expression in a join condition.
- Keep **currency and quantity conversion out of interface views** used at high volume. Convert in the consumption layer where the row count is small.

23.5 The extension cost

When you `extend view entity` on an SAP view, your extension runs for **every consumer of that view**, including SAP's own applications. An extension that traverses an association to a large table can degrade standard Fiori apps that have nothing to do with your requirement.

Before adding a field via extension, ask: does every consumer need to pay for this? If not, model it in your own view instead and join it where you need it.

23.6 Aggregation

Do it on the database:

```
SELECT bukrs, SUM( dmbtr ) AS total
  FROM zdocuments
  WHERE gjahr = @lv_year
  GROUP BY bukrs
  INTO TABLE @DATA(lt_totals).
```

Not by selecting all rows and summing in a loop. This is obvious stated plainly and yet it is what most legacy reports do, because it was reasonable advice on a row-store database twenty years ago.

23.7 Reading a single row

- `SELECT SINGLE` — correct when the full primary key is supplied
- `SELECT ... UP TO 1 ROWS ... ORDER BY` — correct when you want a specific one of several matches
- `SELECT SINGLE` with a partial key returns an arbitrary row. If your code depends on which one, it is already a defect.

23.8 When to use AMDP

Rarely. Justify it. Valid cases:

- A genuinely set-based algorithm CDS cannot express (recursive hierarchies, complex window logic beyond CDS support)
- Very large-volume transformations where the round trip itself dominates

Costs: it is database-specific, harder to test, harder to debug, and in cloud development it is restricted to C4-released artefacts. Most AMDP written in custom code should have been a CDS view.

Chapter 24: ABAP-Side Performance

Once the data is in the application server, the cost profile changes. These are the defects that dominate.

24.1 Choosing the table type

This decision determines the complexity class of every read. Get it wrong and no amount of later tuning helps.

Type	Read by key	Insert	Use when
Standard	Linear scan	Cheap (append)	Sequential processing, small tables, order matters
Sorted	Binary search on key prefix	Ordered insert	Range access, partial key access, sorted output needed
Hashed	Constant time, unique key only	Cheap	Frequent single-row lookup by full unique key

The rule of thumb: if the table is read inside a loop by key, it must be hashed or sorted. A `READ TABLE ... WITH KEY` on a standard table of 50,000 rows inside a loop over 50,000 rows is 2.5 billion comparisons, and it is a weekly occurrence in legacy code.

24.2 Secondary keys

When a table genuinely needs two access paths, define a secondary key rather than maintaining a second table:

```
DATA lt_items TYPE SORTED TABLE OF ty_item
  WITH UNIQUE KEY item_id
  WITH NON-UNIQUE SORTED KEY by_material COMPONENTS material.
```

Costs: memory for the additional index, and maintenance on every write. Use when reads dominate writes, which is the usual case in reporting logic.

24.3 Nested loops

The classic defect:

```
LOOP AT lt_headers INTO DATA(ls_h).
  LOOP AT lt_items INTO DATA(ls_i) WHERE header_id = ls_h-id.
    ...
  ENDLOOP.
ENDLOOP.
```

If `lt_items` is a standard table, the inner `WHERE` is a full scan per outer row. Fixes:

1. Make `lt_items` **sorted by** `header_id` — the runtime can then restrict the inner loop to the matching range.
2. Or use a **parallel cursor**: sort both, keep an index into the inner table and advance it.
3. Or restructure the read so the database returns the joined result and no nesting is required. Usually the right answer.

24.4 Avoid copying rows

`INTO` copies the row. For wide structures inside a large loop, that copy is the dominant cost.

```
" copies each row
LOOP AT lt_data INTO DATA(ls_row).

" no copy
LOOP AT lt_data ASSIGNING FIELD-SYMBOL(<row>).

" no copy, reference semantics
LOOP AT lt_data REFERENCE INTO DATA(lr_row).
```

Use `ASSIGNING` by default in loops over anything non-trivial. The only reason to use `INTO` is when you genuinely need an independent copy to modify.

24.5 Memory discipline

- **Never read an unbounded result set into memory.** Always a filter or package-size processing.
- `FREE` releases the memory; `CLEAR` on an internal table also releases it, but `FREE` is explicit about intent. Use it on large tables you are finished with in long-running code.
- Watch deep structures and nested tables — memory cost multiplies invisibly.
- In mass processing, memory grows across the run unless you actively manage it. A job that works on 10,000 records and dies on 500,000 is almost always a memory defect, not a runtime one.

24.6 Package-size processing

The pattern for genuinely large volumes:

```
SELECT field_a, field_b FROM ztable
  WHERE ...
  INTO TABLE @DATA(lt_pack) PACKAGE SIZE 10000.

  " process lt_pack
  " commit if appropriate
  FREE lt_pack.
ENDSELECT.
```

Package size is a tuning parameter: too small means excessive round trips, too large means memory pressure. Start around 5,000–20,000 rows and measure.

24.7 Strings and conversions

- Repeated string concatenation in a loop creates a new string each time. Build into a table and concatenate once, or use a string buffer approach.
- Implicit type conversions in loops are a silent cost. Declare types that match.
- String templates are readable and generally efficient; the cost is in repetition, not in the construct.

Chapter 25: RAP, OData and UI Performance

The layer where users actually experience slowness, and the layer most often tuned last.

25.1 Determinations and validations

The single biggest RAP performance lever is **trigger scope**.

- A determination `on modify` runs when the triggering fields change. A determination `on save` runs during save. Choose deliberately; do not default to `on save` for everything.
- Scope the trigger to the **specific fields** that matter, not to the whole entity.
- **Never re-read the entire business object** inside a determination. Read only the instances passed to it.
- Determinations that call external services during the interaction phase make every keystroke round trip. Move them to save, or make them asynchronous.

25.2 `$expand` and payload size

Each `$expand` level multiplies the result. A list of 100 orders expanding items expanding schedule lines can produce tens of thousands of rows for a screen that shows twenty.

- Control expand depth in the service and the UI annotations
- Do not expose associations on a list view that the list does not display
- Use `$select` to restrict fields

25.3 Server-side paging

Every list must page. An unbounded `GET` on an entity set is a defect regardless of current data volume, because current data volume is temporary. Confirm `$top` and `$skip` are honoured and that the UI requests them.

25.4 Value helps

Value help views are read constantly and are frequently the slowest thing in an application. Rules:

- A value help view returns few columns
- It filters by the context available at the point of use
- It is not a consumption view built for a different purpose and reused
- Text and description joins are limited to what the help actually displays

25.5 Draft handling

Draft doubles your persistence — every change writes to a draft table before the active table. It is required for genuine draft scenarios and it is overhead everywhere else. Do not enable it by default because a template did.

25.6 Analytical versus transactional

If the requirement is aggregation over large volumes, it belongs in an analytical query consumed by an analytical Fiori app, not in a transactional service that reads rows and sums them. Serving analytics through a transactional RAP service is a design defect that

no amount of tuning will fix, and it is common because the transactional path is the more familiar one.

25.7 Batch and round trips

OData V4 batching reduces round trips for multi-operation interactions. Perceived performance in a Fiori app is dominated by round trip count more often than by server time — measure both before deciding what to fix.

Chapter 26: Measurement and the Tuning Process

26.1 The rule

Never optimise without measurement. Developer intuition about where time goes is wrong most of the time, and "optimised" code that was not measured is usually just less readable code with the same runtime.

26.2 Tools available in ABAP Cloud

Classic transaction-based tools (ST05, SAT, SE30, ST12) are unavailable in Public Cloud and being displaced in ADT everywhere. The cloud-era toolset:

Tool	Location	Answers
ABAP Profiler / ABAP Trace	ADT — Profiling perspective	Where is the time going, by call hierarchy
SQL Trace	ADT	What statements executed, how often, how long
CDS Data Preview	ADT, on a view	Does it return what I expect, at what cost
Explain / dependency analysis	ADT, on a CDS view	What the view actually compiles to
ABAP Unit with runtime measurement	ADT	Regression detection on performance
ATC performance checks	ADT / central ATC	Static detection of known anti-patterns
Application Job monitoring	Fiori	Real runtimes of scheduled work

Confirm availability against the specific release — the ADT tooling in this area has moved substantially over recent releases, and Public Cloud exposes a narrower set than Private Cloud.

26.3 Static detection first

ATC catches a useful subset before anything runs: `SELECT` in loop, `SELECT *`, missing `WHERE`, nested loops on internal tables, `FOR ALL ENTRIES` without an initial check. Enable the performance check group in `Z_CLOUD_STRICT` and make it a gate. It is cheap and it catches the highest-frequency defects.

It does **not** catch: bad table type choice, deep CDS stacks, over-broad determination triggers, or anything volume-dependent. Static analysis is a floor, not a ceiling.

26.4 The tuning loop

1. **Reproduce with production-like volume.** This is non-negotiable and it is where most teams fail. Code that is fast on 500 test records and dies on 5 million is not a tuning problem; it is a test data problem.
2. **Measure.** Profiler for where the time goes; SQL trace for what the database did.
3. **Identify the single largest cost.** Not the most offensive-looking code — the largest measured cost.
4. **Form a hypothesis** about why it is expensive.
5. **Change one thing.**
6. **Re-measure.** Same data, same conditions.
7. **Stop when it is fast enough.** Define “fast enough” before starting, from the business requirement, not from aesthetics.

Record each iteration. A tuning exercise with no record produces no transferable knowledge and cannot be defended when someone asks why the code looks the way it does.

26.5 Designing for performance up front

Cheaper than tuning, and in a cloud world often the only option, because the tuning escape hatches are gone.

At design time, for every data access, answer:

- **What is the expected volume today, and in five years?** Get a number from the business, not a guess.
- **What is the selectivity of the filter?** Reading 10 rows from 10 million is fine; reading 5 million is a different design.
- **How often does this run?** Once a month or once per keystroke.
- **What is the acceptable response time?** From the business requirement.
- **Is this transactional or analytical?** They get different architectures.
- **Where does the calculation belong** on the code-to-data ladder?

A design review that asks these six questions catches more performance defects than any amount of downstream profiling.

26.6 Performance in the definition of done

Add to the code review checklist in Chapter 33:

- ☐ Volume assumption stated and validated with the business
- ☐ Tested against production-like data volume
- ☐ Profiler run on the main path, result recorded
- ☐ No ATC performance findings
- ☐ Response time measured against the stated requirement
- ☐ Table types justified against access patterns
- ☐ For UI: round trip count and payload size checked

26.7 Performance in remediation specifically

Remediation is a performance risk in both directions and both need managing:

The regression risk. Replacing a direct table read with a released CDS view can be slower, because the view carries joins and conversions the raw read did not. Measure the replacement, not just its correctness. Add runtime to the semantic equivalence comparison in Chapter 27's micro-process — same inputs, same outputs, *and* comparable runtime.

The opportunity. Most legacy custom code was written for a row-store database and does the work in ABAP that HANA would do better. Remediation is the one moment when someone is authorised to rewrite it. A remediation programme that reports runtime improvements alongside compliance improvements is a programme with a much easier funding conversation.

Part VI — Brownfield Remediation

This is where the money is and where the expertise is scarcest. Everything before this chapter is preparation.

Chapter 27: The End-to-End Migration Process

The runbook. Ten phases, each with entry criteria, activities, outputs, exit criteria and the way it typically fails. The following three chapters detail the analysis, triage and remediation mechanics referenced here.

27.0 The phase map

```
0  Prerequisites        → readiness gate
1  Discovery            → what exists, what runs
2  Analysis             → what is non-compliant
3  Triage               → what happens to each object
4  Planning             → waves, effort, sequence
5  Foundation           → packages, governance, gates
6  Retire wave          → delete the dead estate
7  Remediation waves    → the core work, by domain
8  Redesign wave        → Dynpro and architecture rebuilds
9  Cutover & hypercare  → go live, stabilise
10 Sustain              → keep it clean
```

Phases 6, 7 and 8 overlap. Phases 0–5 must complete in order, and the most common programme failure is starting phase 7 before phase 5.

Phase 0 — Prerequisites

Objective. Establish that the programme can be executed at all.

Activities. Work Chapter 5 end to end. Confirm the target deployment model in writing. Enable usage logging if not already running.

Exit criteria. The go/no-go checklist in 5.6 is green, or every unticked item is a risk register entry with an owner and a date.

How it fails. Usage logging is not on and nobody notices until triage. The retire case then rests on opinion and the business refuses to sign deletions. This single omission has killed more clean core business cases than any technical issue.

Phase 1 — Discovery

Objective. Know what exists and what actually runs.

Inputs. Repository access, usage logs, where-used index.

Activities. 1. Extract the full custom object inventory: name, type, package, author, last changed, transport history. 2. Extract usage over the agreed window. State the window length in every subsequent document. 3. Build the where-used graph — which objects call which. 4. Map objects to functional areas. Package structure is a weak proxy; expect manual work here. 5. Include **key user extensions** in scope: custom fields, custom logic, custom CDS views, custom business objects. Almost every assessment misses these. 6. Identify the business owner per functional area.

Outputs. The object dataset from Chapter 28.5, populated except for findings and disposition.

Exit criteria. Every object has a functional area and a named owner, or is explicitly flagged as unattributable.

How it fails. Scope is set to "ABAP objects" and the key user layer, the integration layer and the analytical layer are silently excluded. The client discovers the gap at the worst moment.

Phase 2 — Analysis

Objective. Determine technical non-compliance per object.

Activities. 1. Run **S/4HANA readiness** checks — simplification items, field length, removed objects. Relevant to the conversion. 2. Run **ABAP Cloud readiness** checks — released API compliance, forbidden statements. Relevant to clean core. 3. Keep these two result sets **separate**. Clients conflate them constantly and a merged report is unusable. 4. Categorise every finding by remediation pattern (Chapter 30), not just by check ID. Twenty findings of one pattern is one fix applied twenty times. 5. Merge findings into the object dataset.

Outputs. Object dataset with findings, finding counts by priority, and pattern distribution.

Exit criteria. Pattern distribution known. This is what drives estimation.

How it fails. The team reports 40,000 findings and everyone panics. Raw finding counts are meaningless without the pattern distribution and the usage overlay. Never present a finding count on its own.

Phase 3 — Triage

Objective. Assign exactly one disposition to every object.

Activities. 1. Apply the deterministic rules first: zero executions in window → RETIRE candidate. Automate this; it is not a judgement call. 2. Apply assisted classification to the remainder (Chapter 31). 3. **Workshop the RETIRE and REPLACE candidates with business owners.** This is the phase that cannot be compressed or automated. 4. Obtain written sign-off for every RETIRE. 5. For REPLACE, confirm the standard functionality actually covers the requirement — do not assume from the description. 6. For WRAP, record the replacement condition. 7. For FREEZE, record the review date.

Outputs. The disposition register. This is the central artefact of the whole programme.

Exit criteria. 100% of objects have a disposition and a rationale. Zero objects marked "to be decided."

How it fails. Two ways. First, a residual "unknown" bucket that never resolves and quietly becomes FREEZE by default. Second, deletion on usage data alone without business sign-off — which works fine until the annual process runs.

Phase 4 — Planning

Objective. Convert dispositions into a sequenced, estimated programme.

Activities. 1. Estimate by pattern-instance and object type, not by finding count. 2. Apply the testing multiplier explicitly — 40–60% of remediation effort. 3. Group into waves by **functional domain**, not by object type. A wave should be independently testable by one business area. 4. Sequence: retire → quick wins → core domains by business criticality → redesigns. 5. Identify inter-wave dependencies from the where-used graph. 6. Define the wave exit criteria up front.

Outputs. Effort model, wave plan, roadmap.

Exit criteria. Every wave has a scope, an estimate, a test owner and an exit criterion.

How it fails. Waves grouped by object type ("all the reports," "all the BAdIs"). Nothing is independently testable, so nothing can be signed off until everything is done.

Phase 5 — Foundation

Objective. Make it impossible to add new debt while removing old debt.

Activities. 1. Create the package structure (Chapter 13), with cloud language version on everything except `ZCC_WRAP`. 2. Publish the development standards (Chapter 33). 3. Configure ATC check variants and **activate the transport release gate**. 4. Define the exception process and open the register. 5. Define the tier 2 wrapper policy, including a cap. 6. Build the semantic equivalence test harness and prove it on one object. 7. Set up abapGit and the pipeline. 8. Train the development team. Half a day on language version and release contracts prevents months of rework.

Outputs. A development environment where the compliant path is the default path.

Exit criteria. A developer attempting non-compliant code is blocked by the syntax check, not by a reviewer's memory.

How it fails. Skipped, because remediation feels more urgent. The team then remediates 500 objects while writing 200 new non-compliant ones. This is the most expensive mistake in the entire process and it is entirely avoidable.

Phase 6 — Retire wave

Objective. Remove the dead estate. Highest value, lowest risk, fastest visible progress.

Activities. 1. Confirm sign-offs are in place. No sign-off, no deletion. 2. Check the where-used graph for each object — a "dead" object called by a live one is not dead. 3. Deactivate before deleting. Leave deactivated for at least one full business cycle including period end. 4. Delete, in transportable units, with a documented rollback. 5. Record everything. You will be asked about a specific object in two years.

Exit criteria. Objects removed, no incidents attributable to removal after one full period-end cycle.

How it fails. Deletion straight to production without a deactivation period, and a quarter-end job fails. One such incident stops the entire retire wave and often the programme.

Phase 7 — Remediation waves

Objective. Rebuild the objects worth keeping, compliantly.

The per-object micro-process. This is the loop the team runs hundreds of times. Standardise it.

1. Read the disposition and its rationale.
2. Confirm current behaviour with the business owner. Not from documentation — from someone who uses it.
3. **Write characterisation tests against the existing behaviour first.** This is the step teams skip and the reason remediation produces regressions.
4. Identify the released replacement for every non-compliant dependency.
5. **Verify semantic equivalence of every data source swap** against production-like data. Released CDS views commonly apply filters and conversions the raw table does not.
6. Rebuild in a cloud package. Where a non-compliant hook is unavoidable, implement the hook thin and delegate to a tier 1 class.
7. If a wrapper is needed, register it with a replacement condition.
8. Write ABAP Unit tests on the new implementation.
9. Run ATC against `Z_CLOUD_STRICT`.
10. Run the equivalence harness. Old versus new, same inputs, diff outputs.
11. Regression test every consumer from the where-used graph.
12. Transport. Update the disposition register to closed.

Exit criteria per wave. All objects in scope closed, ATC clean, equivalence verified, business area sign-off obtained.

How it fails. Step 5 skipped. The new CDS view excludes cancelled documents and a month-end report silently understates by 3%. Nobody notices for two periods. This is the characteristic failure of assisted remediation specifically, because the tooling makes the swap look trivial.

Phase 8 — Redesign wave

Objective. Rebuild what cannot be remediated in place — chiefly Dynpro programs and architecturally obsolete constructs.

Activities. Treat each as a small project: requirements re-elicitation, UX design, RAP model, Fiori app, full test cycle.

Exit criteria. New app live, old transaction removed, users trained.

How it fails. Estimated as remediation rather than as a project. A 3,000-line Dynpro program with complex screen logic is not a two-day job. Refusing to under-quote this is where you earn credibility with the delivery lead, even though it makes your number look worse.

Phase 9 — Cutover and hypercare

Activities. Sequence the transports respecting dependencies. Run a production-parallel comparison where feasible — old and new paths on the same data, diffed. Monitor the specific objects changed, not just system health. Keep the equivalence harness available during hypercare so a reported discrepancy can be diagnosed in minutes.

Exit criteria. One full period-end closed with no attributable defects.

Phase 10 — Sustain

Objective. Stop the debt from returning. This phase has no end date.

Activities. - ATC gate remains active. Permanently. - Exception register reviewed quarterly; rising exemptions mean the gate is being routed around. - Wrapper register reviewed quarterly against newly released APIs — SAP releases more each release, and each one lets you delete a wrapper. - FREEZE objects revisited at their review dates and at every upgrade. - Custom object count reported monthly. If it is rising, the governance is decorative.

How it fails. The programme ends, the consultants leave, the gate is disabled during a go-live crunch “temporarily,” and it is never re-enabled. Build the gate ownership into a named internal role before you leave.

Chapter 28: Inventory and Analysis

28.1 The sequence

Do not start with ATC. Start with usage.

1. **Collect the object inventory** — every custom object in the system
2. **Collect usage data** — what actually executed, over a long enough window
3. **Run the technical analysis** — ATC cloud readiness / custom code checks
4. **Merge** — technical findings against usage, per object

5. **Triage** — decide the disposition of every object
6. **Estimate** — effort per disposition class
7. **Present** — with a recommendation, not just a report

The order matters because **usage data changes the economics more than technical findings do.** An object with 400 findings that nobody has executed in three years costs nothing to remediate: you delete it.

28.2 Usage data

Sources, in order of usefulness:

- **SAP Usage and Procedure Logging (UPL) / SCMON** — the primary source. Needs to have been running. If it has not been, turn it on and wait.
- **ST03N workload statistics** — coarser but available historically
- **Application-specific logs**

Window length is a judgement call. Twelve months minimum, because of quarter-end, year-end and annual processes. A programme that only runs in March will look dead in a six-month window, and deleting it is a career event. State the window explicitly in every report.

28.3 The technical analysis

Two families of check:

- **S/4HANA readiness checks** — simplification items, field length changes, removed tables and functions. Relevant to the conversion itself.
- **ABAP Cloud readiness checks** — released API compliance, forbidden statements, language version violations.

These are different questions and clients conflate them constantly. A system can be S/4HANA-ready and completely cloud-unready. Separate the two in your reporting.

Run the analysis **remotely** where possible — a central ATC system checking the ECC or S/4 system — so you do not need to install anything in production.

28.4 The Custom Code Migration app

The Fiori app consolidates usage data with findings and produces scoped analyses. It is the fastest route to a first-pass picture. Its output is a starting point, not an answer: it does not know your client's business, and it cannot tell you that three of the four sales order enhancements were superseded in 2021.

28.5 What to record per object

Build this as your working dataset. Everything downstream keys off it.

Field	Source	Why
Object name, type, package	Repository	Identity
Author, last changed date	Repository	Ownership, staleness
Executions in window	UPL/SCMON	The single most important field
Distinct users	UPL/SCMON	Distinguishes batch from real usage
Finding count by priority	ATC	Effort proxy
Finding categories	ATC	Remediation pattern selection
Called-by / calls	Where-used	Blast radius
Business process	Manual / SME	Required for the retire decision
Business owner	Manual	Required for sign-off
Disposition	You	The output

The last three cannot be automated and are exactly why this work is billable.

Chapter 29: The Triage Framework

Every custom object gets exactly one of five dispositions. Force the choice — “we’ll look at it later” is how these programmes stall.

29.1 The five dispositions

RETIRE. No executions in the window, or superseded by standard functionality, or a duplicate. Delete it. *Typical share: 30–60% of the estate.* This is the finding that funds the programme.

REPLACE. The requirement is now covered by standard S/4HANA functionality or a Fiori app. Retire the custom object, adopt the standard. *Typical share: 5–15%.* Requires functional knowledge, which is why technical- only assessments undervalue it.

REMEDIATE. Genuinely needed, and can be rebuilt tier 1. Do the work. *Typical share: 15–30%.*

WRAP. Genuinely needed, no released API exists. Isolate the non-compliant dependency in a tier 2 wrapper and keep the logic tier 1. *Typical share: 5–15%. If it exceeds 20%, your analysis is wrong or the client has an unusually exotic landscape.*

FREEZE. Needed, non-compliant, not worth remediating now. Documented, inventoried, change-frozen, and revisited at the next upgrade. *Typical share: 10–25%.*

29.2 The decision tree

```
Executed in window?
├── No  → Confirm with business owner → RETIRE
└── Yes
    ├── Standard functionality now covers it?
    │   └── Yes → REPLACE
    └── No
        ├── All dependencies released or replaceable?
        │   └── Yes → REMEDIATE
        └── No
            ├── Is the object business-critical or being changed anyway?
            │   ├── Yes → WRAP
            │   └── No  → FREEZE
```

29.3 Rules that keep the triage honest

- **Retire requires a named business owner's sign-off**, in writing. Never delete on usage data alone. This protects the client and protects you.
- **Freeze requires a review date.** A freeze without a date is abandonment.
- **Wrap requires a named replacement condition.** "When SAP releases X, this wrapper is removed." Otherwise the tier 2 layer becomes permanent.
- **Remediate estimates must be by pattern, not by finding count.** Twenty findings of one pattern is one fix applied twenty times.

29.4 Estimation

Do not estimate per finding. Estimate per pattern-instance, then apply a complexity multiplier by object type.

Rough baselines for planning, to be calibrated against the first sprint:

Disposition	Effort per object
Retire	0.25 day (including sign-off admin)
Replace	1–3 days (mostly functional analysis and testing)
Remediate, simple	0.5–2 days
Remediate, complex	3–10 days
Wrap	1–3 days
Freeze	0.25 day (documentation only)

Testing dominates. Budget testing at 40–60% of total remediation effort. A programme that budgets 10% for testing is a programme that will slip, and you should say so before it starts rather than after.

Chapter 30: Remediation Pattern Catalogue

These are the findings you will see repeatedly. Learn the fix for each; this catalogue is what makes remediation fast.

30.1 Direct table access

Finding: `SELECT ... FROM <SAP table>` where the table is not released.

Fix hierarchy:

1. Find the released CDS view over the same data. Most core business tables now have an `I_*` interface view.
2. If the released view exists but lacks a field, check whether the field is on a released association.
3. If no released view exists, tier 2 wrapper exposing exactly the needed projection.

Watch for: semantic differences. Released CDS views frequently apply filters and conversions the raw table does not — client, deletion indicators, currency conversion, language. Compare result sets before you swap, always. This is the most common source of silent regressions in remediation work.

30.2 Unreleased function module calls

Finding: `CALL FUNCTION 'SOME_SAP_FM'`.

Fix hierarchy:

1. Look for a released class-based API covering the same capability. Many classic FMs now have `CL_*` or `XCO_*` equivalents.
2. Look for a released BAPI. BAPIs are frequently C1 or C2 released.
3. Check whether a RAP business object covers the transaction.
4. Tier 2 wrapper as last resort.

Watch for: update task semantics. A wrapper around an update FM must preserve the LUW behaviour or you will corrupt data under load.

30.3 Modifications and enhancement implementations

Finding: modification to SAP code, or an implicit/explicit enhancement.

Fix hierarchy:

1. Is there a released BAdI or RAP behaviour extension for the same purpose? This is the ideal outcome and more common than people expect.
2. Is the requirement now standard configuration?
3. If the only hook is classic: implement the classic hook and **delegate immediately to a tier 1 class**. Two lines in the hook, all logic in cloud code.

Modifications specifically: check whether the modification is still needed at all. A large share of ECC modifications addressed bugs that SAP fixed years ago, or gaps that S/4HANA closed.

30.4 Obsolete language constructs

Finding: header lines, `TABLES`, short-form statements, `MOVE`, obsolete system fields, implicit conversions.

Fix: mechanical. This is the category most amenable to automated and AI-assisted rewriting, and where you get the fastest visible progress. Do it first for programme momentum, but do not confuse it with real remediation — it is cosmetic relative to dependency violations.

30.5 Dynpro and GUI-based UI

Finding: classic Dynpro, selection screens, ALV GUI, `WRITE` lists.

Fix: this is a redesign, not a remediation. Options:

- Rebuild as a Fiori elements app over a RAP service (preferred for transactional)
- Rebuild as an analytical Fiori app over a CDS query (preferred for reporting)
- Retire and use a standard app

Estimate honestly. A 3,000-line Dynpro program with complex screen logic is not a two-day job. This category is where remediation programmes lose their schedule, and where you earn credibility by refusing to under-quote it.

30.6 File and OS access

Finding: `OPEN DATASET`, `GUI_UPLOAD` / `GUI_DOWNLOAD`, `CALL 'SYSTEM'`.

Fix: redesign to released alternatives — application server file APIs where released, or move the file handling entirely outside the core to an integration layer. In Public Cloud there is no filesystem; the integration layer is the only answer.

30.7 Native SQL and AMDP

Finding: `EXEC SQL`, `ADBC`, AMDP over unreleased tables.

Fix: rewrite in Open SQL or CDS. AMDP is permitted in cloud development but only against C4-released database artefacts. Most hand-written AMDP over SAP tables fails this.

30.8 Custom tables

Finding: custom tables are usually fine. The issue is exposure.

Fix: release your own tables and views with C1 so that cloud packages may consume them, and expose them through a CDS interface view rather than letting consumers select from the table directly. This is good practice regardless of clean core.

Chapter 31: AI-Assisted Remediation

This is your differentiator. Almost nobody has both the release contract depth and the tooling instinct. Here is how to do it without producing garbage.

31.1 Where AI genuinely helps

Triage classification. Given an object's findings, usage, where-used graph and source, propose a disposition with reasoning. This is a judgement task with a well-defined output space and abundant signal — a good fit. Human confirms.

Pattern-based rewriting. Obsolete constructs, header lines, `SELECT`-to-CDS swaps where the mapping is known. High volume, low ambiguity.

Where-used and impact summarisation. Turning a 400-row where-used list into "this is called from three batch jobs and one Fiori app, all in the order-to- cash stream."

Documentation generation. Every frozen and wrapped object needs documentation. Nobody writes it. AI writes a decent first draft from the source and the findings.

Released API discovery. "What is the released equivalent of `BAPI_X`?" is a retrieval question over a large, structured, poorly-indexed corpus.

Test case generation. First-pass ABAP Unit tests around code you are about to change — genuinely useful, because the absence of tests is the main reason remediation is slow.

31.2 Where AI will hurt you

Semantic equivalence. An LLM will confidently swap a table read for a CDS view that applies a different filter. It does not know the view excludes cancelled documents. **Every data-source swap needs a result set comparison against production-like data.** No exceptions.

LUW and update task semantics. Models reason poorly about ABAP's update task, `COMMIT WORK` behaviour and lock semantics. Human review mandatory.

Business logic intent. AI cannot tell you the 2009 workaround exists because of a specific customer contract. Only an SME can.

Release state. Model training data is stale on release contracts. Never trust an assertion that an object is released — verify in the target system, every time.

Authorisation logic. Never let generated code decide an authority check.

31.3 A workable architecture

```
ATC findings (XML/API)  ─┐
UPL usage extract       ─┼→  Normalised object dataset
Where-used graph        ─┤
Source code             ─┘
                              ↓
                     Triage engine (LLM + rules)
                              ↓
               Proposed disposition + reasoning + confidence
                              ↓
                     Human review queue (sorted by confidence)
                              ↓
               Approved dispositions → remediation backlog
                              ↓
          Pattern-based generation → ADT → ATC gate → test → transport
```

Design principles:

- Rules first, model second. Anything deterministic (zero executions → RETIRE candidate) should be a rule, not a prompt. Cheaper, faster, auditable.
- Emit **confidence**, and sort the human review queue by it. Low-confidence items get the expensive human attention.
- Emit **reasoning**, always. A disposition without a rationale cannot be defended in a steering committee.
- Never auto-apply. Generate, review, apply.
- Log every decision. You will be asked why object X was deleted, eighteen months later.

31.4 The governance question clients will ask

“Are you sending our source code to an LLM?” Have an answer before the meeting. Options: on-premise or VPC-hosted models, SAP’s own AI services, contractual data-processing terms, or restricting the AI layer to metadata and findings rather than source. Know which the client’s security team will accept. This question kills more AI-assisted proposals than technical failure does.

Part VII — Governance and Delivery

Chapter 32: Landscape, Transport and Lifecycle

An architect who cannot answer “how do we develop through the upgrade” is not an architect. This chapter is the operational half of the role.

32.1 Landscape design

Minimum viable: DEV → QA → PRD. What is usually also needed:

System	Purpose	Commonly omitted, and it hurts
Sandbox	Throwaway experimentation, POCs	Yes — so experiments happen in DEV
DEV	Development, unit tests	No
QA	Integration and regression testing, production-like data	Data volume is the omission, not the system
PRD	Production	No
N+1 / project track	Parallel development during a conversion or upgrade	Yes — and this is the expensive one

32.2 The dual-track problem

During a conversion or a major upgrade you have two truths at once: production needs fixes, and the project is building the future state. Two tracks, and every fix applied to production must also reach the project track.

Options, with honest costs:

- **Retrofit.** Fixes go to production and are re-applied to the project track, manually or with tool support. Cost: continuous effort, and drift when someone forgets.
- **Freeze.** No production changes during the project. Cost: rarely survives contact with a real business, and pretending otherwise sets the programme up to fail.
- **Single track with feature toggles.** New code ships dormant. Cost: toggle complexity and discipline, but no retrofit.

Decide this before the project starts. A retrofit process invented in month four, after divergence has already occurred, is a reconciliation exercise.

32.3 Source control and branching

abapGit is the practical mechanism for source control of custom development.

A workable model:

- `main` reflects production

- `develop` is the integration branch
- Feature branches per work item, short-lived
- Release branches when a project track exists

The honest limitation: abapGit gives you the source, not the transport. Objects still move through the transport system, and the mapping between a merged branch and a released transport must be managed deliberately. gCTS narrows this gap where the client has adopted it. Do not present abapGit as if it made ABAP work like a normal git project — it does not, and overselling it loses you credibility with anyone who has used it.

32.4 Transport strategy

- Transports sequenced by dependency, not by readiness. A transport released before its prerequisite produces an import error at the worst moment.
- **Transport of copies** for testing without committing to a release.
- The ATC gate on release (Chapter 34) — the only gate that reliably works, because it is unavoidable.
- Object locking discipline: two developers, one object, one transport.
- A documented rollback for anything structural.

32.5 BTP lifecycle, where side-by-side is in scope

- Separate subaccounts for dev, test and production. Not separate spaces in one subaccount.
- Deployment through a CI/CD pipeline, not manual deployment from a laptop.
- Cloud Transport Management for promoting between subaccounts.
- **Coordination is the hard part**: an on-stack service change and its BTP consumer must be released together, or the consumer must tolerate both versions. Design for the second — a synchronised release across two lifecycles is fragile and will eventually fail.

32.6 Absorbing upgrades

Public Cloud — quarterly, not optional.

- Know the release calendar and the test window
- Regression suite must run inside that window, which means automated
- Read the release notes for deprecations affecting your extensions
- Extensions break on removed released APIs — rare, but it happens, and the deprecation notice is your warning

Private Cloud — you control timing, which is a trap.

The freedom to defer becomes years of deferral, and then a conversion-sized upgrade. Set a maximum age policy and hold to it.

Either way, the point of clean core is that upgrade absorption becomes routine. If it is not routine, the extensions are not compliant, whatever the ATC report says. That is the real test, and it happens once a quarter.

32.7 Feature toggles

Useful for: dual-track development, phased rollout, decoupling deployment from release, and emergency disablement of new logic without a transport.

Costs: every toggle is a branch in the code and a combination to test. Set an expiry on each one and remove it. A codebase with forty permanent toggles is worse than one with none.

32.8 Hotfix path

Define it before you need it: who approves, which systems it traverses, whether it bypasses the ATC gate (it should not), how it is reconciled with the project track, and how it is regression tested. A hotfix path invented during an incident is how the gate gets disabled permanently.

32.9 Lifecycle review checklist

- ☐ Landscape includes a sandbox and, if converting, a project track
- ☐ Dual-track strategy decided before the project starts
- ☐ Branching model agreed and documented
- ☐ Transport sequencing and rollback documented
- ☐ BTP subaccounts separated by stage
- ☐ Cross-lifecycle release coordination designed for tolerance, not synchrony
- ☐ Upgrade regression suite automated and inside the test window
- ☐ Maximum system age policy set (Private Cloud)
- ☐ Feature toggles have expiry dates
- ☐ Hotfix path documented, and it does not bypass the gate

Chapter 33: Development Guidelines

A remediation programme without governance re-creates the mess within two years. This chapter is the standards document you hand to a client’s development team.

33.1 The non-negotiables

1. All new development uses **ABAP for Cloud Development**. No exceptions granted by developers; exceptions granted only by the architect, in writing, with a review date.

2. Standard ABAP exists in exactly **one package** (`ZCC_WRAP`). Any new Standard ABAP package requires architect approval.
3. Every tier 2 wrapper has a documented justification and a replacement condition.
4. No object is transported without a clean ATC run against the agreed variant.
5. No modification to SAP objects. Ever. Not “rarely” — never. Once you permit one, you permit the category.
6. All CDS consumption goes through interface views, never direct table access, even for custom tables.
7. UI annotations live in metadata extensions, not inline.
8. Every new business object exposes a business event where another system might plausibly care.

33.2 The exception process

You need one, or people will simply ignore rule 1.

- Requested by the developer, in the transport request description or a tracked ticket
- Approved by the architect
- Recorded in a central register with: object, reason, replacement condition, review date
- Reviewed quarterly

An exception register with fewer than five entries after a year means either excellent design or an unenforced process. Check which.

33.3 Code review checklist

- Is the object in the correct package with the correct language version?
- Does it consume only released APIs, or does it route through `ZCC_WRAP`?
- Does any wrapper leak an unreleased type in its signature?
- Are CDS extensions cardinality-preserving?
- Are there ABAP Unit tests, and do they use test doubles rather than production data?
- Are UI annotations in a metadata extension?
- Is a business event published where relevant?
- Is the object documented well enough that someone can decide its disposition in five years?

Chapter 34: Quality Gates and Pipelines

34.1 ATC as a gate, not a report

An ATC report that runs monthly and is emailed to a distribution list changes nothing. Make it a gate:

- **In ADT** — developers run ATC locally before release. Configure the default check variant so this is the path of least resistance.

- **On transport release** — block release on priority 1 and 2 findings. This is the gate that actually works, because it is unavoidable.
- **Centrally, scheduled** — trend reporting for management, and to catch objects that were exempted.

34.2 Check variants

Maintain at least two:

- `Z_CLOUD_STRICT` — for new development. Full cloud readiness, naming conventions, security checks, performance checks.
- `Z_REMEDIATION` — for the brownfield estate. Focused on the findings you are actually remediating this phase, so the report is actionable rather than overwhelming.

A single variant producing 40,000 findings is a variant nobody uses.

34.3 Exemptions

ATC exemptions are a governance surface, not a convenience. Require a reason, an approver and an expiry. Report on exemption count as a programme metric — rising exemptions mean the gate is being routed around.

34.4 Transport and CI/CD

- **abapGit** for source control of custom development. Non-negotiable for anything you want to reuse, review properly, or move between landscapes.
- **gCTS** where the client has adopted it, for git-based transport.
- **Pipeline stages:** syntax → ATC → ABAP Unit → transport → integration test.
- **Test automation** is the constraint on remediation velocity. A client with no ABAP Unit coverage will remediate at a third of the speed of one with it, and will find defects in production instead of in test. Raise this in week one; it is the highest-leverage thing a programme can fix early.

Chapter 35: Metrics

Report these. They make progress visible and they protect you when the programme is questioned.

Volume metrics

- Total custom objects, trending down
- Objects by disposition (retired / replaced / remediated / wrapped / frozen)
- Percentage of estate assessed

Compliance metrics

- Objects in cloud-compliant packages, as a share of active objects
- Priority 1 and 2 ATC findings, trending down
- Tier 2 wrapper count — **should plateau then decline**, never grow linearly

- Active exemptions, with age

Quality metrics

- ABAP Unit coverage on remediated objects
- Defects found in test vs production, per remediated object
- Transport rejection rate at the ATC gate

Business metrics

- Estimated upgrade effort avoided
- Custom object count reduction as a maintenance cost proxy
- Objects blocking a Public Cloud move, trending to zero

That last one is the one executives care about, because it is the one tied to a strategic option. Lead with it.

Chapter 36: Running a Client Assessment

The packaged service. This is what you sell.

36.1 A four-week assessment

Week 1 — Scope and collect. Landscape and release inventory. Confirm target deployment model — this changes every recommendation. Enable UPL if not running. Extract object inventory. Set up remote ATC. Identify business owners per functional area.

Week 2 — Analyse. Run readiness checks. Merge findings with usage. Build the where-used graph. First-pass automated triage. Identify the pattern distribution.

Week 3 — Validate. Workshops with functional owners on the RETIRE and REPLACE candidates. This is the week that requires a human and cannot be compressed. Confirm the business-critical set. Sample-verify the automated triage.

Week 4 — Recommend. Disposition per object. Effort model. Phased roadmap. Governance proposal. Executive readout.

36.2 What the deliverable contains

1. **Executive summary** — one page. Estate size, disposition split, effort range, the strategic option it preserves.
2. **Current state** — inventory, usage, findings, by functional area.
3. **Disposition register** — every object, its disposition, its rationale. This is the substance and it is what they are paying for.
4. **Effort model** — by phase, by disposition class, with the testing assumption stated explicitly.
5. **Roadmap** — phased, with a quick-win phase first.
6. **Governance proposal** — the standards, the gate, the exception process.
7. **Risk register** — including the objects you could not classify and why.

36.3 Sequencing the programme

Order the phases for momentum and cash:

1. **Retire** — highest value, lowest risk, visible immediately. Fund the rest from the savings narrative.
2. **Governance in place** — stop the bleeding before repairing.
3. **Quick-win remediation** — mechanical patterns, high volume, low risk.
4. **Core remediation** — the real work, by business domain.
5. **Redesigns** — Dynpro rebuilds and similar, last, because they are projects in their own right.

36.4 The three questions that win the meeting

Have these ready:

- *"Which of your five clean core dimensions are you actually addressing?"* Almost always the answer is "extensions only," and pointing at data and integration reframes the conversation immediately.
- *"How many of your custom objects were executed in the last twelve months?"* Most clients do not know. The number is always lower than they expect.
- *"If you had to move to Public Cloud in three years, what stops you today?"* This converts a technical debt conversation into a strategic one, which is where budget lives.

Part VIII — The Specialist Path

Chapter 37: The Twelve-Month Plan

37.1 The position

Not “clean core consultant” — the topic is crowded. The defensible position is narrower:

> ***Custom code remediation and clean core extensibility for S/4HANA brownfield conversions, executed with AI-assisted triage at scale.***

The moat is the combination. Plenty of people understand release contracts. Plenty of people build AI tooling. Very few have both, and the second half is worth little without the first — an AI triage tool built by someone who does not understand C0 versus C1 produces confident nonsense.

37.2 Months 1–3: Access and grounding

- **Get staffed on a brownfield conversion or remediation stream.** This is the single highest-value action available and everything else is secondary to it. If your current employer has no such pipeline, that is a serious signal about staying.
- Set up a system you can actually work in. Given hardware constraints, BTP ABAP Environment trial or a client system, not a local container.
- Work the whole release contract catalogue until API State reasoning is automatic.
- Build the object dataset schema from Chapter 28 for a real system.

37.3 Months 4–6: Method

- Triage a real estate end to end. Even a few hundred objects.
- Build the remediation pattern catalogue *for that client’s actual findings* — your Chapter 30 with real examples.
- Build the first version of the triage tool. Rules only, no model. Prove the data pipeline works.
- Start publishing. Weekly, narrow, specific.

37.4 Months 7–9: Leverage

- Add the model layer to the triage tool. Confidence scoring, reasoning output, review queue.
- Validate its dispositions against your own manual triage. Measure agreement. That measurement is the credibility artefact.
- Write up the methodology properly.

37.5 Months 10–12: Position

- Package the four-week assessment as a defined offering with a fixed scope and price.
- Speak somewhere. A user group, an SAP community event, a webinar.
- Convert the published work into inbound.

37.6 What to publish

Specific beats broad, every time. Not “AI in SAP” — that space is saturated with people who have never remediated anything.

Publish instead:

- Individual remediation patterns with before and after code
- “We found 3,200 objects and retired 1,900 — here is how the triage worked”
- The C0/C1/C2 confusion, explained properly, with examples
- Honest assessments of where AI failed in remediation and why
- The tier 2 wrapper pattern, with the governance rules

The failure-mode posts are the ones that build authority. Everyone posts successes. The person who publishes “here is where the model swapped a table for a CDS view with different semantics and how we caught it” is obviously someone who has done the work.

Chapter 38: Traps

Certification is a weak signal here. A live cutover on the CV is what the market prices. Get the certification if a client demands it as a procurement tick, not as a learning strategy.

Do not become the tool person. The tool is leverage for your judgement. If you are known as “the guy with the script,” you get paid like a script. Lead with the method, mention the tooling second.

Do not over-promise automation. Triage is assisted, not automated. The moment a client believes remediation is automatic, your value evaporates and your first semantic regression becomes a scandal instead of a caught defect.

Do not confuse ATC-clean with correct. Code can pass every check and be wrong. Semantic equivalence testing is the actual quality gate.

Do not let the tier 2 layer grow. Track it as a metric from day one. A wrapper layer nobody caps becomes the new modification layer within two years, and you will have built the thing you were hired to remove.

Do not spread. The single biggest risk to this plan is starting a second thing at month four. Depth and reputation are both compounding assets and both reset to zero when abandoned. One bet, twelve months, nothing new started until it is done.

Appendix A: Pre-Development Checklist

Before writing any new object:

- ☐ Correct package, correct software component, correct language version
- ☐ Every SAP dependency checked for API State in the *target* system
- ☐ Extension point confirmed as C0 released if extending
- ☐ Key user extensibility ruled out with a reason
- ☐ Side-by-side ruled out with a reason
- ☐ Naming convention followed
- ☐ Test strategy decided before coding
- ☐ Business event considered

Appendix B: Remediation Object Checklist

For each object being remediated:

- ☐ Usage confirmed and business owner identified
- ☐ Disposition recorded with rationale
- ☐ All findings categorised by pattern
- ☐ Released replacement identified for each dependency
- ☐ **Semantic equivalence verified against production-like data**
- ☐ LUW and update task behaviour preserved
- ☐ Authorisation checks reviewed by a human
- ☐ ABAP Unit tests written
- ☐ ATC clean against the agreed variant
- ☐ Where-used consumers regression tested
- ☐ Documentation updated

Appendix C: Tier 2 Wrapper Register Template

Field	Content
Wrapper object	`ZCC_WRAP_*`
Capability wrapped	
SAP object depended on	
Why no released API	
Replacement condition	
Owner	
Created	
Review date	
Consumers	

Appendix D: Glossary

ADT — ABAP Development Tools, the Eclipse-based IDE. **AMDP** — ABAP Managed Database Procedure. **ATC** — ABAP Test Cockpit, the static analysis framework. **BAdI** — Business Add-In, SAP's enhancement hook mechanism. **C0–C4** — release contracts governing stability guarantees. **CDS** — Core Data Services, the declarative data modelling layer. **Developer extensibility** — on-stack custom development using ABAP Cloud. **Key user extensibility** — no-code/low-code extension via Fiori apps. **LUW** — Logical Unit of Work, the transactional boundary. **RAP** — ABAP RESTful Application Programming Model. **Side-by-side extensibility** — extensions running on BTP, outside the core. **Tier 1/2/3** — SAP's extensibility tier model. **UPL / SCMON** — usage logging, the source of execution data. **VDM** — Virtual Data Model, SAP's layered CDS view conventions. **Code-to-data** — moving computation to the database rather than moving data to the application server. **Column store** — HANA's default storage, optimised for reading few columns across many rows. **Package size** — processing a large result set in bounded chunks. **Parallel cursor** — a loop technique that avoids re-scanning an inner table. **Secondary key** — an additional index on an internal table.

Appendix E: Performance Review Checklist

Condensed from Chapter 18. Use in code review.

Database - [ ] No `SELECT` in a loop - [ ] No `SELECT *` - [ ] Filtering and aggregation on the database - [ ] `FOR ALL ENTRIES` driver checked for initial - [ ] Round trips minimised

CDS - [ ] Stack depth justified - [ ] Associations preferred to joins for optional fields - [ ] Cardinality declared - [ ] Filters pushed down - [ ] No conversion in high-volume interface views - [ ] Extension cost to other consumers considered

Internal tables - [ ] Table type matches access pattern - [ ] No key read on a large standard table inside a loop - [ ] No nested loop over two large tables - [ ] `ASSIGNING` used in loops over wide rows

Memory - [ ] No unbounded result set - [ ] Package-size processing for mass volumes - [ ] Large tables freed when finished

RAP / UI - [ ] Determination triggers scoped to fields - [ ] No whole-BO reads in determinations - [ ] `$expand` depth controlled - [ ] Server-side paging active - [ ] Value help views lean - [ ] Draft enabled only where needed - [ ] Analytics served by an analytical query

Process - [ ] Volume assumption stated - [ ] Tested at production-like volume - [ ] Profiler run and result recorded - [ ] Measured against a stated response-time requirement

Appendix F: Architecture Decision Record Template

One per significant decision. Half a page. The value is that it exists.

Field	Content
ID / date	
Decision	One sentence, in the active voice
Status	Proposed / Accepted / Superseded by
Context	The requirement and the constraints, including the numbers
Options considered	Each with its main trade-off
Decision and rationale	Why this one, in terms of the constraints above
Consequences	What this makes easy, what it makes hard, what it forecloses
Compliance impact	Tier, released APIs used, wrapper introduced (if any)
Review trigger	The event that would make us revisit this

The test of a good ADR: someone joining in two years can read it and understand why the system looks the way it does, without asking anyone.

Appendix G: Clean Core Rule Card

One page. Cite by ID in code review and governance documents. Full text in Chapter 9.

Architecture — CC-A1 no modifications · CC-A2 released APIs only · CC-A3 target confirmed in writing · CC-A4 rejected options recorded · CC-A5 hooks delegate, logic stays tier 1 · CC-A6 no database-level integration

Development — CC-D1 cloud language version · CC-D2 one Standard ABAP package · CC-D3 wrappers justified and dated · CC-D4 no transport with P1/P2 findings · CC-D5 CDS views not direct table select · CC-D6 annotations in metadata extensions · CC-D7 tests run without a populated system · CC-D8 volume stated and tested

Governance — CC-G1 architect-only exceptions · CC-G2 quarterly exception review · CC-G3 wrapper register reviewed each release · CC-G4 object count reported monthly · CC-G5 gate ownership named internally

Data and integration — CC-I1 every interface has an owner · CC-I2 idempotent async consumers · CC-I3 versioned contracts · CC-I4 personal data annotated

Adoption order: CC-A3 → CC-D1/CC-D2 → CC-D4 → CC-G1 → the rest.

Closing Note

The technical content here has a shelf life. Release contracts get extended, new APIs get published, ATC variants get renamed, Fiori apps get consolidated. Verify against the target system before you present anything to a client.

What does not have a shelf life is the method: inventory before analysis, usage before findings, disposition before estimation, governance before remediation, and semantic verification before transport. That sequence is what separates a remediation programme that finishes from one that becomes a permanent cost centre.

The rest is doing it on a real system, repeatedly, until the judgement is yours rather than borrowed from a manual.