# ABAP Test Cockpit (ATC)

## Preface

ABAP Test Cockpit (ATC) is not just a syntax checker or a code inspector. It's the **central nervous system for ABAP quality governance** across your entire landscape — on-premise or cloud. The real maturity of an ABAP team is reflected in how they use ATC — not as a compliance gate, but as a continuous feedback and improvement mechanism.

Let's walk through ATC from the ground up — not in textbook terms, but as it works in real projects, with examples and practical insights.

## 1. The Core Idea of ATC – "Your ABAP Quality Gatekeeper"

Think of ATC as your **customs officer** in a busy airport.
Every piece of ABAP code that wants to "go live" must pass through a scanner.
This scanner checks for performance risks, security violations, syntax mismatches, and non-compliance with modern ABAP principles.

Earlier, we used manual code reviews and Code Inspector. Now, ATC brings all that under one **automated and centralized governance framework**.

## 2. ATC Layers – How It Actually Works in Projects

| Layer | Description | Example |
|---|---|---|
| **Local Check Layer** | Developers run ATC locally before transport release. | Dev runs `/ATC` to check syntax, performance hints, unused variables. |
| **Central Check Layer** | A quality system validates code from multiple systems. | ECC and S/4HANA devs send code to Central ATC in QAS. |
| **Governance Layer** | COE defines global check variants and waivers. | "No direct SELECTs on MARA in Cloud" rule applied globally. |

This layered setup helps maintain **local agility with global discipline**.

## 3. Real-Time Scenarios

## Scenario 1: S/4HANA Readiness Check

When moving from ECC to S/4HANA, ATC identifies:

- Deprecated tables (e.g., BSEG, VBFA)
- Obsolete FMs and structures
- SQL patterns not optimized for HANA

**View:**
An SME adjusts the variant to skip trivial warnings and focus on high-impact findings — like table accesses that should move to CDS.
`SELECT * FROM MARA` becomes a CDS view call with `@AbapCatalog.buffering`.

---

## Scenario 2: Pre-Transport Quality Enforcement

Developers often rush transports before EOD.
Integrating ATC at transport release ensures:

- Transports with critical errors are blocked.
- Warnings may pass but get logged.
- Failures route to a "code approver."

**Tip:**
Central ATC + Transport Exit = Automated governance.
No manual policing — just disciplined automation.

---

## Scenario 3: Governance Across Landscapes

You may have multiple systems — ECC, S/4, BTP ABAP.
By using a **Central ATC**, one ruleset governs all, ensuring **cross-system consistency**.

**Example:**
"No direct DB access" enforced across all environments.
One central variant = uniform coding standard.

---

## Scenario 4: Custom Rules (The Real Power)

Beyond SAP's standard checks, you can create **custom ATC checks** using `CL_CI_CHECKBASE`.

Examples:

- Block hardcoded company codes

- Enforce CDS view usage instead of SELECTs
- Require `@AccessControl.authorizationCheck`

**Use:**
Log violations into a Fiori dashboard for developer trend tracking.
That's continuous improvement made visual.

---

## 4. How ATC Differs Between On-Premise and Cloud

| Feature | On-Premise | ABAP Cloud (Steampunk/S/4HANA Cloud) |
|---|---|---|
| Ruleset | Flexible | Fixed, SAP-governed |
| Exemptions | Allowed via waiver process | Restricted |
| Custom Checks | Possible | Not allowed |
| Central Check System | Optional | Mandatory |

**Takeaway:**
On-prem, *you* are the gatekeeper.
In Cloud, **SAP** is.
So, ATC mastery is critical in hybrid models.

---

## 5. Exemption Handling (Waivers Done Right)

Sometimes findings can't be fixed immediately.
You can request **justified exemptions** with defined validity.

Example:
"FM `REUSE_ALV_GRID_DISPLAY` will be replaced post go-live."
Approved for 3 months, auto-rechecked later.

**Best Practice:**
Track expiry, justification, and approval.
Avoid long-living waivers — they become technical debt.

---

## 6. ATC in Continuous Integration Pipelines

Modern teams embed ATC into:

- Jenkins pipelines

- gCTS workflows
- GitHub Actions for ABAP Cloud

When a transport is committed, ATC runs automatically.
Only if checks pass, deployment continues.
That's **DevOps discipline inside ABAP**.

---

## 7. Common Real-World Misuses

| Misuse | Impact | Solution |
|---|---|---|
| Treating ATC like a syntax check | Misses design flaws | Define proper variants |
| Ignoring findings | Recurring issues | Enforce checks pre-transport |
| Using standard checks only | No business fit | Add custom checks |
| Too many exemptions | Governance loss | Track and expire waivers |

---

## 8. ABAP Cloud Readiness with ATC

In ABAP Cloud (Steampunk/S/4 Cloud), ATC enforces:

- Use of released APIs only
- No direct DB access
- Clean code and exception handling

**Takeaway:**
ATC here is **SAP's policy engine** — pass it, or you can't deploy.

---

## 9. SME Tips for ATC Maturity

1. Begin with awareness before enforcement.
2. Customize variants by project.
3. Automate transport-level blocking.
4. Manage waivers properly.
5. Analyze findings over time.
6. Integrate ATC in DevOps flows.
7. Teach "fixing mindset," not "passing mindset."

---

## 10. Closing Thought

ATC is not a tool — it's a **culture of quality**.
It transforms firefighting into foresight.

From dummy to expert, your real growth lies in understanding **why those rules exist** and how to evolve with them.

---

# 11. Top Usages of ATC and Its Variants

Variants define the **purpose** of ATC in your ecosystem.
Each one is a different lens — compliance, performance, modernization, or governance.
Below are the top variants that matter in real-world ABAP landscapes.

---

### 1. S/4HANA Readiness Variant

**Purpose:** Identify legacy code incompatible with S/4 data models.
**Used In:** ECC-to-S/4HANA migrations.
**Checks:** Obsolete tables, deprecated FMs, outdated SQL patterns.

**Real Use:**
Mapping old SELECTs on VBAK/VBAP to CDS views like `I_SalesDocument`.
Iterative runs show progress as you modernize.

**Takeaway:**
Run it iteratively — it's your modernization speedometer.

---

### 2. Performance and Security Variant

**Purpose:** Optimize runtime efficiency and security posture.
**Used In:** Performance tuning, audits.
**Checks:** Nested SELECTs, missing indexes, SQL injection risks.

**Real Use:**
A 6-hour batch job optimized to 8 minutes after fixing SELECT loops flagged by ATC.

**Takeaway:**
Even with HANA, inefficient SQL costs you. Use this variant to catch it early.

---

### 3. Code Quality and Naming Convention Variant

**Purpose:** Enforce consistent standards across teams.
**Used In:** Global rollouts with mixed vendor teams.
**Checks:** Naming patterns, obsolete keywords, comment headers.

**Real Use:**
Custom check blocked direct SELECTs on MARA to enforce CDS-only access.

**Takeaway:**
Consistency isn't cosmetic — it's future-proofing.

---

## 4. ABAP Cloud and Clean Code Variant

**Purpose:** Validate Cloud compliance and API whitelisting.
**Used In:** Steampunk, BTP, S/4HANA Cloud.
**Checks:** Released APIs, clean exception handling, no GUI classes.

**Real Use:**
Detected forbidden `CL_GUI_FRONTEND_SERVICES` class in BTP app — replaced with `/UI2/CL_JSON`.

**Takeaway:**
ATC is your **Cloud compliance radar**. Use it pre-deployment.

---

## 5. Security and GDPR Compliance Variant

**Purpose:** Prevent data exposure and legal breaches.
**Used In:** Banking, Healthcare, HR systems.
**Checks:** Hardcoded credentials, unmasked PII, missing `AUTHORITY-CHECK`.

**Real Use:**
Logging user emails flagged as GDPR breach. Replaced with anonymized GUIDs.

**Takeaway:**
This isn't about code quality — it's about trust.

---

## 6. Custom Governance Variant (Homegrown Rules)

**Purpose:** Enforce organization-specific standards.
**Used In:** COE-driven landscapes.
**Checks:** Custom code structure, naming, CDS-only data access.

**Real Use:**
A global client's "ZCOE_CLEAN_CODE" variant enforced 12 internal rules — reduced violations by 70%.

**Takeaway:**
Make ATC reflect your internal architecture philosophy.

---

## 7. Regression and Release Validation Variant

**Purpose:** Catch reintroduced issues automatically.
**Used In:** CI/CD or release gates.
**Checks:** Compares deltas and validates fixed issues.

**Real Use:**
A reverted SELECT * caught during pipeline run — prevented rollback of quality.

**Takeaway:**
Regression variants make quality continuous, not reactive.

---

## 8. Custom Performance Baseline Variant

**Purpose:** Track performance posture over time.
**Used In:** Long-term tuning programs.
**Checks:** SQL complexity, join depth, buffer usage.

**Real Use:**
Performance dashboards built on ATC metrics became monthly KPIs.

**Takeaway:**
Transform ATC from gatekeeper to teacher.

---

## 9. Integration with Code Review Workflows

**Purpose:** Automate pre-review insights.
**Used In:** Agile and DevOps models.
**Checks:** Runs automatically and attaches findings to review notes.

**Real Use:**
ATC findings linked to Jira saved 40% of review time.
Reviewers focus on logic, not syntax.

**Takeaway:**
ATC refines peer review; it doesn’t replace it.

---

## 10. Legacy System Stabilization Variant

**Purpose:** Bring legacy code under control.
**Used In:** ECC landscapes with thousands of Z-objects.
**Checks:** Obsolete statements, redundant programs, outdated tables.

**Real Use:**
ATC identified 1,200 redundant objects — upgrade time dropped by 25%.

**Takeaway:**
ATC is your **digital janitor** — cleaning decades of debt silently.

---

## 12. Closing Insight: The Real Role of Variants

Each variant is a **different mirror** to your codebase:

- Performance lens
- Security lens
- Clean code lens
- Compliance lens
- Governance lens

The true expert doesn't run one variant once.
They **embed ATC across every lifecycle stage** — from design to deployment.

When used right, ATC stops being a static report and becomes a **living ecosystem of quality** — one that evolves as your ABAP landscape evolves.

---

**Final Takeaway:**
If Code Inspector was the "spell-check," ATC is the "grammar engine" of ABAP quality.
It teaches, enforces, and scales good habits across people, systems, and generations of code.

---

# 13. Practical Guide: Configuring an ATC Variant (Step-by-Step with Example)

---

## Purpose of This Section

This guide walks you through **how to configure your own ATC variant**, why each step matters, and how to align it with real-time project goals.

We'll take a working example — creating a **"ZCOE_PERFORMANCE_AND_CLEAN_CODE"** variant — that enforces both **performance efficiency** and **clean ABAP standards**.

---

# 1. Prerequisites and Context

Before creating a variant, make sure you understand:

- You have **authorization** to create or modify Code Inspector variants (transaction **SCI**).
- You know the **target scope** — whether the variant will be used **locally** by developers or **centrally** in ATC governance.
- You've decided **which checks** matter for your project phase (e.g., performance, S/4 readiness, naming conventions, Cloud compliance).

**Real-World Context:**
In a large S/4HANA implementation, the COE typically owns a **Central Check System**, and developers in satellite systems only "consume" these variants during pre-transport validation.

---

# 2. Step-by-Step Configuration

## Step 1: Go to Transaction `SCI` (Code Inspector)

Even though ATC is the "new face" of code quality, it internally uses **Code Inspector variants** as its foundation.
Hence, creating or customizing a variant starts in `SCI`.

- Execute transaction `SCI`
- Click **"Management of Inspection Variants"**
- Choose **Create**

Enter:

- **Variant Name:** `ZCOE_PERFORMANCE_AND_CLEAN_CODE`
- **Short Text:** "Performance and Clean Code Rules for COE"

**Tip:** Always prefix custom variants with `Z` or `Y` and maintain naming consistency — it helps identify ownership during ATC synchronization.

---

## Step 2: Select the Check Categories

You'll see multiple check categories like:

- **Syntax Check**
- **Extended Program Check**
- **Performance Checks**
- **Security Checks**
- **Naming Conventions**
- **S/4HANA Readiness Checks (if applicable)**

For our example, select:

- **Extended Program Check**
- **Performance Checks**
- **Security Checks**
- **Naming Conventions**

---

## Step 3: Choose Specific Checks Within Each Category

Let's tailor the checks.
Click each category and **select sub-checks** relevant to your use case.

### *Performance Checks*

- "SELECT * without WHERE clause"
- "SELECT inside LOOP"
- "Unbuffered access to buffered tables"
- "Nested SELECTs"
- "Missing ORDER BY with SELECT DISTINCT"

### *Security Checks*

- "Missing AUTHORITY-CHECK"
- "Dynamic SQL without ESCAPE usage"
- "Hardcoded credentials or URLs"

### *Naming Conventions*

- "Object naming does not follow ZCL_/ZIF_ pattern"
- "Method naming not in CamelCase"
- "Comment header missing"

### *Extended Program Check*

- "Obsolete statements (MOVE-CORRESPONDING, TABLES, etc.)"
- "Obsolete function modules (like `REUSE_ALV_GRID_DISPLAY`)"

**Out-of-Box Tip:**
If your project is on **ABAP Cloud or RAP**, include "Use of unreleased APIs" and "Forbidden statements in Cloud environment."

## Step 4: Adjust the Check Parameters

Each check can be fine-tuned.
For instance:

- For "SELECT * without WHERE clause", you can specify **table exceptions** (e.g., small customizing tables like T001 or T005).
- For "Missing AUTHORITY-CHECK", you can **whitelist** certain programs where user authorization is handled externally.
- For "Naming conventions", you can define your **pattern rules** (e.g., classes must start with `ZCL_` and not `ZCLASS_`).

**Expert Recommendation:**
Keep initial rules tight but fair — don't overwhelm developers with thousands of warnings in early phases.
Gradually enforce more categories as the project matures.

---

## Step 5: Save and Activate the Variant

Once your rules are selected and configured:

- Click **Save**
- Assign to a transport (if it needs to move to QAS or Central System)
- Make sure you set the **variant as "Active"**

Now your variant exists as a Code Inspector variant, which ATC can consume.

---

# 3. Linking the Variant to ATC

Now move to **transaction ATC → Setup → Runs → Check Variant Assignment**.

Here you can:

- Assign your variant to a **Check Run Series** (like "Daily Nightly ATC" or "On Transport Release")
- Define **Scope** (Program, Package, Transport)
- Specify **System Role** (Local or Central)

## Example Setup:

| Parameter | Value |
|---|---|
| **Variant** | `ZCOE_PERFORMANCE_AND_CLEAN_CODE` |

| Parameter | Value |
|---|---|
| **Check Scope** | Transported Objects Only |
| **Check Mode** | Remote (if using Central ATC System) |
| **Execution Schedule** | Nightly 2:00 AM |
| **Result Retention** | 10 Days |

**Practical Insight:**
This setup runs ATC nightly and stores results for 10 days.
Devs get automated quality reports each morning, similar to "SonarQube dashboards" in Java environments.

---

## 4. Running and Testing the Variant

Run transaction `/ATC` → **"Check Objects"**

You can choose:

- Program
- Package
- Transport Request
- Entire custom namespace (`Z*`)

Select your variant `ZCOE_PERFORMANCE_AND_CLEAN_CODE` and **Execute**.

You'll get a structured results tree:

- **Category** → Performance, Security, Clean Code
- **Severity** → Error, Warning, Information

Double-click any issue → takes you to the code line directly.

---

## 5. Refining and Reusing Variants

After running for a few days, review findings with the team:

- Remove noise (false positives)
- Add new checks as your codebase stabilizes
- Save it as a new version: `ZCOE_PERFORMANCE_V2`

You can export/import variants between systems using:

- **Program:** `RS_CODE_INSPECTOR_VARIANTS_EXPORT`
- **Program:** `RS_CODE_INSPECTOR_VARIANTS_IMPORT`

This allows you to replicate governance across multiple systems.

---

## 6. Governance Layer: Approval and Waiver Workflow

If a developer disagrees with a finding (e.g., justified performance exception), they can request a **waiver**.
In Central ATC, you can:

- Approve, reject, or set expiry for exemptions.
- Maintain audit trails.

**Example:**
Report `ZSD_TOP_CUSTOMERS` flagged for "SELECT *".
Developer justifies that CDS migration is planned in the next release.
Approver grants 3-month waiver.
After expiry, ATC rechecks automatically — sustainable and trackable governance.

---

## 7. Practical Example Summary

| Step | Action | Example |
|---|---|---|
| 1 | Create Variant in `SCI` | `ZCOE_PERFORMANCE_AND_CLEAN_CODE` |
| 2 | Choose Categories | Performance, Security, Naming |
| 3 | Configure Checks | "SELECT inside LOOP", "AUTHORITY-CHECK missing" |
| 4 | Save Variant | Assign to transport |
| 5 | Link in ATC Setup | Schedule nightly or on transport release |
| 6 | Test Run | Run `/ATC` → Results |
| 7 | Refine & Govern | Waiver control + Trend monitoring |

---

## 8. Expert-Level Customization: Building Your Own Check

For deeper control, you can **create a custom ATC check class**.
Example — disallow hardcoded company codes.

1. Create a new class `ZCL_ATC_CHECK_CC` inheriting from `CL_CI_TEST_BASE`.
2. Implement method `IF_CI_TEST~RUN`.
3. Parse code for hardcoded literals like `'1000'`.
4. Raise a message using `add_message`.

Register this class under **Custom Checks** in `SCI` → Add to your variant.

This allows you to enforce rules SAP never imagined — your own coding DNA.

---

## 9. Real-Time Governance Example

In one S/4HANA implementation, the COE created three ATC variants:

- `ZCOE_READINESS` → S/4 migration findings
- `ZCOE_PERFORMANCE` → Runtime tuning
- `ZCOE_CLEAN_CORE` → RAP & Cloud readiness

Each developer had to pass all three before transport release.
This policy reduced post-go-live issues by 80%.

---

## 10. Closing Insight

Configuring an ATC variant is **not a one-time setup** — it's a **living governance artifact**.
It evolves with your project maturity and development discipline.
Start small, tune regularly, and integrate deeply.

In a mature setup:

- ATC runs automatically with every transport.
- Developers fix issues before QA finds them.
- Quality is continuous — not an afterthought.

That's when you know your **ATC variant has moved from configuration to culture**.

---