Architecture Decision Record (ADR) Template

What is an ADR?

An Architecture Decision Record documents an important architectural or technical decision made during the project, along with its context and consequences.

ADRs help the team:

- Remember why decisions were made
- Avoid revisiting settled decisions
- Understand the trade-offs that were considered
- Onboard new team members
- Learn from past decisions

ADRs are stored in Git alongside your code because:

- Decisions and code evolve together
- Git history shows when and why decisions changed
- Pull Request discussions become part of the record
- ADRs are versioned just like code
- Anyone with code access can see the architectural decisions

---

When to write an ADR

Write an ADR when:

- Choosing between multiple viable technical alternatives
- Making decisions that are difficult or expensive to change later
- Deviating from established patterns or conventions
- Decisions significantly impact non-functional requirements (performance, security, scalability)
- A decision affects multiple parts of the system

Examples requiring an ADR:

- Choosing a database schema approach
- Selecting an API authentication method
- Deciding on state management strategy (frontend)
- Choosing between REST and GraphQL
- Database migration strategy
- **GDPR compliance approach (data retention, anonymization, user rights)**
- **Privacy by design implementation**

Examples NOT requiring an ADR:

- Variable naming conventions
- Individual component file structure
- Minor refactoring decisions
- Bug fix approaches

When in doubt, ask: "Would future team members need to understand why we made this choice?" If yes, write an ADR.

ADR Process

1. **Create** the ADR file in `/docs/architecture/decisions/` using the format: `ADR-XXX-short-title.md`
2. **Draft** the ADR with status "Proposed" and commit to a feature branch
3. **Discuss** with the team (Daily Scrum, Sprint Planning, or dedicated session)
4. **Review** by at least 2 team members via Pull Request
5. **Update** status to "Accepted" when team reaches consensus
6. **Merge** the ADR into the main/dev branch

Git workflow:

- ADRs are stored in the Git repository alongside code
- Create ADRs in feature branches, just like code
- Use Pull Requests for review and discussion
- Status changes are new commits (preserves history in Git)
- Never delete ADRs - change status to "Superseded" instead

Important: Every status change must be documented in the "Status History" section at the bottom of the ADR, including the date and reason for the change.

ADRs are **immutable** once accepted. If a decision changes, create a new ADR that supersedes the old one.

What "immutable" means:

- The **content** (context, decision, rationale) cannot be edited after acceptance
- The **status** can be updated (Accepted → Superseded or Deprecated)
- Minor corrections (typos, formatting) are allowed, but no changes to meaning

Why immutability matters:

- Preserves historical record of why decisions were made
- Shows what information was available at the time
- Allows learning from past decisions
- Prevents revisionist history

ADR Template

Copy the template below when creating a new ADR:

```
# ADR-XXX: [Decision Title]

**Repository:** [eap-architecture | eap-backend | eap-frontend | eap-qa]
**Date:** YYYY-MM-DD
**Status:** [Proposed | Accepted | Superseded | Deprecated]
**Deciders:** [Names of team members involved]
**Category:** [Platform Decision | Application Decision]

## Context

What is the issue or problem we're trying to solve?

- Describe the forces at play (technical, business, organizational)
- What requirements or constraints exist?
- What's the current situation?

Keep this section factual and neutral.

## Decision

What did we decide to do?

State the decision clearly and concisely. Use active voice:
- "We will use PostgreSQL for data persistence"
- "We will implement authentication using JWT tokens"

## Consequences

What are the results of this decision?

### Positive
- What benefits do we gain?
- What problems does this solve?
- What becomes easier?

### Negative
- What trade-offs are we accepting?
- What becomes harder?
- What risks are we taking on?

### Neutral
- What else changes as a result?
- What future decisions does this constrain?

## Alternatives Considered

What other options did we evaluate?

For each alternative:
1. **[Alternative name]**
   - Description
   - Why we chose not to pursue this
   - Key trade-offs vs. chosen decision
```

```
If no alternatives were seriously considered, explain why this decision was obvious.

## Constraints

List any constraints that limited our options:
- Platform constraints (e.g., "Must use PostgreSQL - platform requirement")
- Time constraints
- Budget constraints
- Technical constraints
- Team skill constraints

## Notes

Any additional context, references, or implementation notes:
- Links to relevant documentation
- Proof of concept results
- Performance benchmarks
- Examples from similar projects

## Related ADRs

### In this repository:
- List ADRs in this repository that are related to this decision

### In other repositories:
- Use full GitHub links when referencing ADRs from other repositories
- Example: [eap-architecture ADR-003: API Authentication](https://github.com/org/eap-
architecture/blob/main/decisions/ADR-003-api-authentication.md)

---

**Last Updated:** YYYY-MM-DD
**Supersedes:** [ADR-XXX] (if applicable)
**Superseded By:** [ADR-XXX] (if applicable)

---

## Status History

Document all status changes here to maintain a complete audit trail:

| Date | Status | Reason |
|------|--------|--------|
| YYYY-MM-DD | Proposed | Initial draft |
| YYYY-MM-DD | Accepted | Team consensus reached in Sprint X Planning |
| YYYY-MM-DD | Superseded | Replaced by ADR-XXX due to [reason] |

---
```

Example ADR

Here's a complete example:

```
# ADR-001: Database Schema Versioning Strategy

**Repository:** eap-backend
**Date:** 2026-01-08
**Status:** Accepted
**Deciders:** Alice (DevOps lead), Bob (Backend dev), Carol (Product Owner)
**Category:** Application Decision

## Context

Our application needs a way to manage database schema changes across development, testing,
and production environments. Multiple developers will be creating database changes
simultaneously across different feature branches.

Requirements:
- Schema changes must be trackable and reversible
- Changes must be applied consistently across environments
- Changes must work with our Git Flow branching strategy
- Solution must integrate with our CI/CD pipeline

## Decision

We will use Alembic (SQLAlchemy migrations) for database schema versioning.

Migration files will be:
- Stored in `/backend/migrations/versions/`
- Generated using `alembic revision --autogenerate`
- Reviewed in pull requests before merging
- Applied automatically by CI/CD pipeline during deployment

## Consequences

### Positive
- Schema changes are version controlled alongside code
- Automatic migration generation reduces manual work
- Built-in rollback capability
- Wide adoption in Python/SQLAlchemy ecosystem
- Good integration with our testing pipeline

### Negative
- Autogenerate doesn't catch all schema changes (indexes, constraints sometimes missed)
- Requires discipline to review generated migrations
- Merge conflicts possible when multiple branches create migrations simultaneously
- Team needs to learn Alembic-specific commands

### Neutral
- Migration files become part of code review process
- Deployments take slightly longer (running migrations)
- Need to establish naming convention for migration files
```

```
## Alternatives Considered

1. **Raw SQL Migration Scripts**
   - Would give us more control over SQL
   - Rejected because: doesn't integrate well with SQLAlchemy ORM, no automatic generation,
more manual work

2. **Django Migrations**
   - Excellent migration system
   - Rejected because: we're using FastAPI, not Django

3. **Flyway**
   - Industry standard for Java/enterprise
   - Rejected because: overkill for our project size, less Python-native

## Constraints

- Must work with PostgreSQL (platform requirement)
- Must work with SQLAlchemy (chosen ORM)
- Must integrate with GitHub Actions (CI/CD platform)

## Notes

- Alembic documentation: https://alembic.sqlalchemy.org/
- Team training session scheduled for Sprint 2
- Initial migration setup tracked in ticket EAP-15

## Related ADRs

### In this repository (eap-backend):
- None yet - this is the first backend-specific ADR

### In other repositories:
- [eap-architecture ADR-002: Technology Stack](https://github.com/org/eap-
architecture/blob/main/decisions/ADR-002-tech-stack.md) - Prescribes PostgreSQL and
SQLAlchemy

---

**Last Updated:** 2026-01-08

---

## Status History

| Date | Status | Reason |
|------|--------|--------|
| 2026-01-07 | Proposed | Initial draft by Alice |
| 2026-01-08 | Accepted | Team consensus in Sprint 1 Planning |

---
```

ADR Numbering

ADRs are numbered sequentially **within each repository**: ADR-001, ADR-002, ADR-003, etc.

Why sequential per repository instead of global numbering or prefixes?

We chose this approach because:

1. **Industry standard** - This is how ADRs are typically numbered in the wider software community
2. **Simple** - No coordination needed across repositories to "claim" the next number
3. **Repository provides natural scope** - The repository name already indicates the scope (platform, backend, frontend, QA)
4. **Explicit cross-references** - When referencing ADRs from other repos, you must include the repository name, making dependencies visible

This means:

- `ADR-001` exists in multiple repositories (eap-architecture, eap-backend, eap-frontend, eap-qa)
- Each repository maintains its own sequential numbering
- Cross-repository references must include the repository name

Example:

- `eap-architecture/decisions/ADR-001-repository-strategy.md`
- `eap-backend/docs/decisions/ADR-001-database-schema.md`
- `eap-frontend/docs/decisions/ADR-001-state-management.md`

These are three different ADRs with the same number but in different scopes.

ADR Organization by Repository

eap-architecture/decisions/

Purpose: Platform-level and cross-cutting decisions

When to create ADRs here:

- Decisions affecting multiple components (backend + frontend + QA)
- Infrastructure and deployment decisions
- API contracts and authentication
- Technology stack choices
- Development workflow decisions

Examples:

- Repository strategy
- Tech stack selection
- API authentication approach
- Deployment strategy

- Git workflow

eap-backend/docs/decisions/

Purpose: Backend-specific implementation decisions

When to create ADRs here:

- Decisions only affecting backend code
- Implementation details within backend constraints
- Backend-specific patterns

Examples:

- Database schema design
- ORM usage patterns
- Caching strategy
- Module structure
- Background job processing

eap-frontend/docs/decisions/

Purpose: Frontend-specific implementation decisions

When to create ADRs here:

- Decisions only affecting frontend code
- Frontend implementation patterns
- UI/UX technical decisions

Examples:

- State management approach
- Component library choice
- Routing strategy
- Form handling
- Client-side caching

eap-qa/docs/decisions/

Purpose: QA and testing-specific decisions

When to create ADRs here:

- Testing strategy decisions
- Test tooling choices
- QA-specific processes

Examples:

- E2E testing framework

- Test data management
- Performance testing approach
- Test environment setup

Cross-Repository References

When referencing an ADR from another repository, **always include the repository name** and ideally provide a full GitHub link.

Good examples:

- "This implements the authentication approach defined in eap-architecture ADR-003"
- "See backend ADR-002 for database migration strategy"
- "Related to eap-frontend ADR-001 (State Management)"

Poor examples:

- "See ADR-003" (Which repository?)
- "As per the authentication ADR" (Which number? Which repo?)

Finding ADRs

To find the next available number in your repository:

1. Check the `/decisions/` or `/docs/decisions/` directory
2. Look at the highest numbered ADR
3. Use the next sequential number

To find all ADRs across the project:

- Each repository's `/decisions/` folder should have a README.md with an index
- eap-architecture/decisions/README.md links to other repository decision directories

---

File Naming Convention

Format: `ADR-XXX-short-descriptive-title.md`

Rules:

- Use three-digit zero-padded numbers: `001`, `002`, `010`, `100`
- Use lowercase with hyphens for the title
- Keep titles concise but descriptive (3-6 words)
- Avoid abbreviations unless widely understood

Good examples:

- `ADR-001-repository-strategy.md`

- `ADR-002-database-schema-versioning.md`
- `ADR-015-frontend-state-management.md`
- `ADR-023-api-authentication-approach.md`

Poor examples:

- `ADR-1-repo-strat.md` (not zero-padded, too abbreviated)
- `ADR-042-we-decided-to-use-postgresql-for-persistence.md` (too long)
- `ADR-008-Database_Schema.md` (mixed case, underscore)

ADR Categories

Platform Decision

Decisions requiring staff review (after team input):

- **Tech stack versions** (React 18 vs 19, Python 3.11 vs 3.12)
- **Ecosystem choices** (component library with niche vs mainstream community)
- Infrastructure choices
- Security implementations
- Deployment strategies
- Tool selections that affect entire system
- **GDPR compliance approach (staff/DPO consulted)**
- **Data retention and privacy policies**
- **Decisions impacting learning goals or market preparation**

Criteria for Platform Decisions:

- Affects entire system or multiple components
- Hard to change later (high switching cost)
- Requires experience/judgment beyond team level
- Impacts learning goals or market preparation
- Team lacks context to evaluate long-term implications

Application Decision

Decisions owned by the team (within platform constraints):

- **Implementation within chosen tech stack**
- **Specific components within approved ecosystem**
- Database schema design (with privacy considerations)
- API endpoint structure
- Frontend component organization
- Implementation patterns

- Code organization and naming conventions
- **Technical implementation of GDPR requirements (anonymization, export, etc.)**

Criteria for Application Decisions:

- Within established platform constraints
- Team can evaluate trade-offs
- Relatively changeable (lower cost to change)
- Good learning opportunity for team
- Team can experiment and learn from mistakes

Grey Area Decision Tree

If unsure which category, ask:

1. **Does this impact learning goals or market preparation?**
   → If YES: Platform Decision
2. **Is this hard/expensive to change later?**
   → If YES: Platform Decision
3. **Does team have enough context to evaluate long-term implications?**
   → If NO: Platform Decision
4. **Can team safely experiment and learn from mistakes?**
   → If NO: Platform Decision
5. **Is this primarily about implementation details within constraints?**
   → If YES: Application Decision

When in doubt: Mark as "Application Decision" and discuss with staff coach during ADR review. Staff will guide categorization.

Tips for Writing Good ADRs

Be specific

- Bad: "We will use a modern frontend framework"
- Good: "We will use React 19 with TypeScript and Zustand for state management"

Explain the 'why', not just the 'what'

- Document the reasoning and context
- Future team members need to understand the forces at play

Example: Platform Decision with Team Input

```
# ADR-003: React Version Selection

## Status
Accepted

## Category
```

```
Platform Decision (Staff decided after team input)

## Context
Team discussed using React v18 vs React v19 for the frontend.

**React v18 Arguments:**
- Team member has prior experience with v18
- Specific component library preference (ecosystem familiarity)
- Proven stability in production environments

**React v19 Arguments:**
- Latest version with performance improvements (concurrent rendering, automatic batching)
- Better preparation for Dutch IT market (industry moving to v19)
- Mainstream ecosystem with stronger European/global community support
- Modern best practices and patterns

## Decision
We will use **React v19** with mainstream component libraries.

## Consequences

**Positive:**
- Team learns latest React features and patterns
- Better alignment with Dutch IT industry trends
- Strong community support and extensive documentation
- Modern performance optimizations built-in

**Negative:**
- Team member's v18 experience less directly applicable
- Slightly newer, less battle-tested in production
- Learning curve for v19-specific features

**Mitigation:**
- Team member with React experience helps team understand core concepts
- React fundamentals (components, hooks, state) remain consistent across versions
- Focus on learning modern patterns over leveraging existing v18 knowledge

## Alternatives Considered
1. **React v18 with team member's preferred library**: Rejected due to niche community and
learning goals prioritization
2. **Vue.js or other framework**: Out of scope - React already chosen as platform

## Constraints
- **Learning Goal:** Prepare trainees for Dutch corporate IT market
- **Community:** Prefer mainstream libraries with large European/global communities
- **Maintenance:** Avoid libraries with small or region-specific communities
- **Market Relevance:** Industry is moving toward v19, not staying on v18

## Deciders
Staff (after team discussion and input)

## Notes
This decision prioritizes learning goals and market preparation over
```

```
leveraging existing team member experience. Team member's React knowledge
remains valuable for teaching fundamental concepts to other team members.

**Key Learning:** Even when team has relevant expertise, platform decisions
may prioritize educational outcomes and industry alignment over immediate
technical convenience.
```

This example demonstrates:

- Platform decision made by staff after team input
- Transparent rationale including trade-offs
- Recognition of team member's input
- Clear connection to learning goals
- Honest acknowledgment of downsides

Keep it concise

- Aim for 1-2 pages maximum
- Link to external resources rather than copying documentation

Use Git effectively

- Meaningful commit messages for ADR updates
- PR discussions capture team reasoning
- Git blame shows who wrote which section
- Git history shows evolution of the decision

Be honest about trade-offs

- Every decision has downsides
- Acknowledging them shows mature thinking

Update status when decisions change

- Don't edit the content of accepted ADRs
- Change status to "Superseded" and link to new ADR
- Create a new ADR that supersedes the old one
- Both ADRs remain as historical record

FAQs

Q: Can we change an accepted ADR?
A: You can update the **status** (e.g., Accepted → Superseded), but you cannot change the **content** (the decision, reasoning, or context). If the decision changes, create a new ADR that supersedes the old one. Both ADRs remain in the repository as a historical record.

Q: How do we document status changes?
A: Add an entry to the "Status History" table at the bottom of the ADR. Include the date, new status, and reason for the change. For example: "2026-02-15 | Superseded | Replaced by ADR-023 due to new security requirements"

Q: What about typos or formatting fixes?
A: Minor corrections that don't change the meaning are acceptable. Use your judgment - if it changes understanding of the decision, create a new ADR instead.

Q: Do ADRs go through code review?
A: Yes! ADRs are reviewed via Pull Requests just like code. The PR discussion becomes part of the decision record. Require at least 2 reviewers.

Q: Should ADRs be in the same repository as the code?
A: Yes. ADRs live in `/docs/architecture/decisions/` in your main code repository. This keeps decisions and code synchronized.

Q: What if we made a decision but didn't write an ADR?
A: Write one retroactively if the decision is still relevant and important.

Q: Why can the same number (e.g., ADR-001) exist in multiple repositories?
A: Each repository maintains its own sequential numbering. The repository name provides the scope. This is simpler than trying to coordinate a global numbering system across all repositories and is the industry standard approach.

Q: How do I reference an ADR from another repository?
A: Always include the repository name and ideally a full GitHub link. Example: "See eap-architecture ADR-003". This makes cross-repository dependencies explicit and visible.

Q: Which repository should my ADR go in?
A: Ask yourself: "Does this decision affect only one component or multiple?" If it affects multiple components (e.g., API design affects both backend and frontend), put it in eap-architecture. If it's specific to one component's internal implementation, put it in that component's repository. See the "ADR Organization by Repository" section for detailed guidance.

Q: Do all team members need to agree?
A: Aim for consensus. If there's strong disagreement, document both positions in the ADR and let the Product Owner make the final call.

Q: How many ADRs should we have?
A: Quality over quantity. For a 12-week project, expect 5-15 ADRs total.

Q: What if the decision is obvious?
A: If it's truly obvious, the ADR will be short. But "obvious" decisions often aren't obvious later or to newcomers.

End of ADR Template