AWS Prescriptive Guidance

Using architectural decision records to streamline technical decision-making for a software development project

AWS Prescriptive Guidance: Using architectural decision records to streamline technical decision-making for a software development project

Table of Contents

Using architectural decision records to streamline technical decision-making for a software development project

Darius Kunce and Dominik Goby, Amazon Web Services (AWS)

March 2022 (document history (p. 12))

This guide introduces the architectural decision records (ADR) process for software engineering projects. ADRs support team alignment, document strategic directions for a project or product, and reduce recurring and time-consuming decision-making efforts.

During project and product development, software engineering teams need to make architectural decisions to reach their goals. These decisions can be technical, such as deciding to use the command query responsibility segregation (CQRS) pattern, or process-related, such as deciding to use the GitFlow workflow to manage source code. Making these decisions is a time-consuming and difficult process. Teams must justify, document, and communicate these decisions to relevant stakeholders.

Three major anti-patterns often emerge when making architectural decisions:

- No decision is made at all, out of fear of making the wrong choice.
- A decision is made without any justification, and people don't understand why it was made. This results in the same topic being discussed multiple times.
- The decision isn't captured in an architectural decision repository, so team members forget or don't know that the decision was made.

These anti-patterns are particularly important to tackle during the development process of a product or project.

Capturing the decision, the context, and considerations that led to the decision in the form of an ADR enables current and future stakeholders to collect information about the decisions made and the thought process behind each decision. This reduces software development time and provides better documentation for future teams.

Targeted business outcomes

ADRs target three business outcomes:

- They align current and future team members.
- They set a strategic direction for the project or product.
- They avoid decision anti-patterns by defining a process to properly document and communicate architectural decisions.

ADRs capture the context of the decision to inform future stakeholders. A collection of ADRs provide a hand-over experience and reference documentation. Team or project members use the ADR collection for follow-up projects and product feature planning. Being able to reference ADRs reduces the time required during development, reviews, and architectural decisions. ADRs also allow other teams to learn from, and gain insights into, considerations made by other project and product development teams.

ADR process

An architectural decision record (ADR) is a document that describes a choice the team makes about a significant aspect of the software architecture they're planning to build. Each ADR describes the architectural decision, its context, and its consequences. ADRs have states and therefore follow a lifecycle. For an example of an ADR, see the appendix (p. 10).

The ADR process outputs a collection of architectural decision records. This collection creates the decision log. The decision log provides the project context as well as detailed implementation and design information. Project members skim the headlines of each ADR to get an overview of the project context. They read the ADRs to dive deep into project implementations and design choices.

When the team accepts an ADR, it becomes immutable. If new insights require a different decision, the team proposes a new ADR. When the team accepts the new ADR, it supersedes the previous ADR.

Scope of the ADR process

Project members should create an ADR for every architecturally significant decision that affects the software project or product, including the following (Richards and Ford 2020 (p. 9)):

- Structure (for example, patterns such as microservices)
- Non-functional requirements (security, high availability, and fault tolerance)
- Dependencies (coupling of components)
- Interfaces (APIs and published contracts)
- Construction techniques (libraries, frameworks, tools, and processes)

Functional and non-functional requirements are the most common inputs to the ADR process.

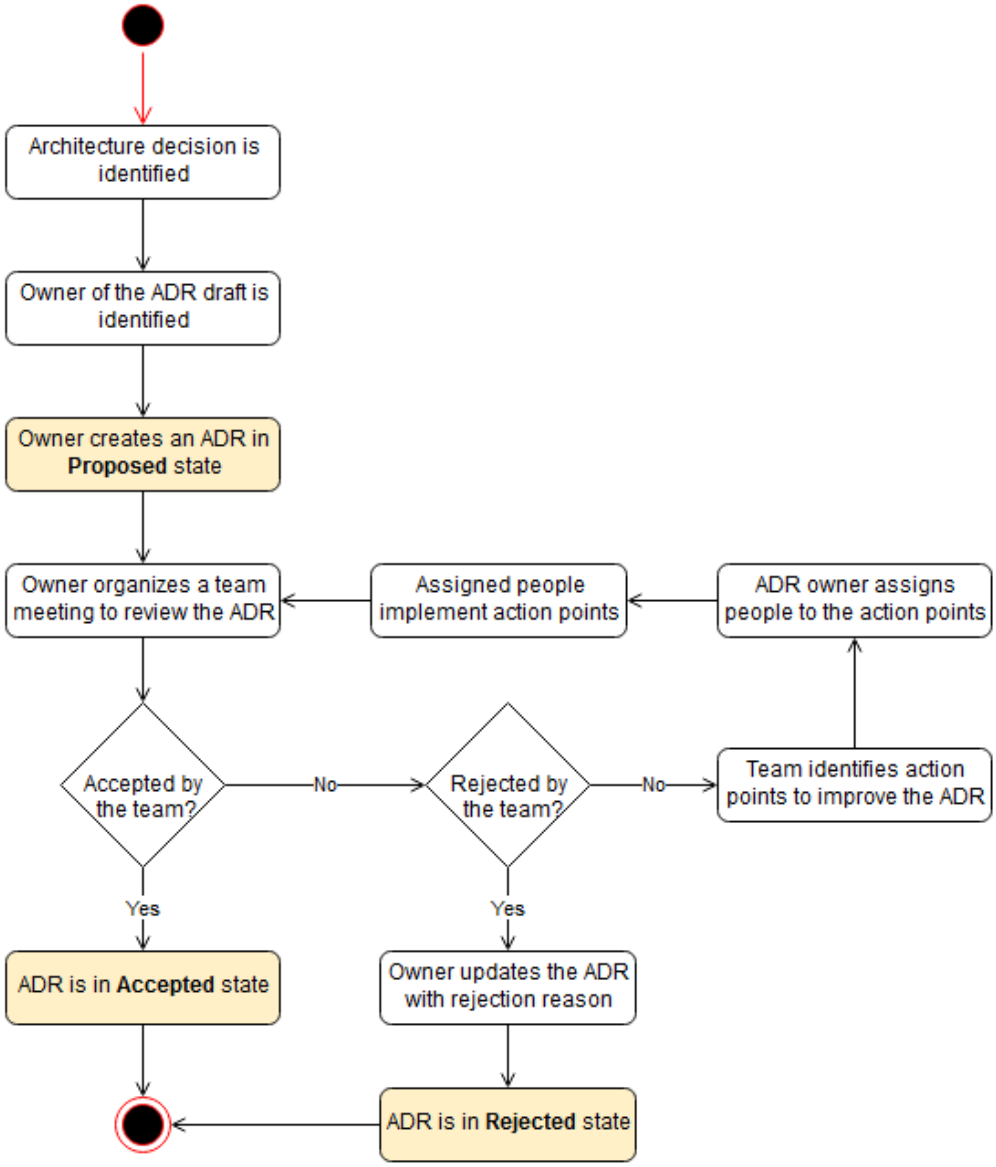
ADR contents

When the team identifies a need for an ADR, a team member starts to write the ADR based on a projectwide template. (See the ADR GitHub organization for example templates.) The template simplifies ADR creation and ensures that the ADR captures all the relevant information. At a minimum, each ADR should define the context of the decision, the decision itself, and the consequences of the decision for the project and its deliverables. (For examples of these sections, see the appendix (p. 10).) One of the most powerful aspects of the ADR structure is that it focuses on the reason for the decision rather than how the team implemented it. Understanding why the team made the decision makes it easier for other team members to adopt the decision, and prevents other architects who weren't involved in the decision-making process to overrule that decision in the future.

ADR adoption process

Every team member can create an ADR, but the team should establish a definition of ownership for an ADR. Each author who is the owner of an ADR should actively maintain and communicate the ADR content. To clarify this ownership, this guide refers to ADR authors as *ADR owners* in the following sections. Other team members can always contribute to an ADR. If the content of an ADR changes before the team accepts the ADR, the owner should approve these changes.

The following diagram illustrates the ADR creation, ownership, and adoption process.



After the team identifies an architectural decision and its owner, the ADR owner provides the ADR in the **Proposed** state at the beginning of the process. ADRs in the **Proposed** state are ready for review.

The ADR owner then initiates the review process for the ADR. The goal of the ADR review process is to decide whether the team accepts the ADR, determines that it needs rework, or rejects the ADR. The project team, including the owner, reviews the ADR. The review meeting should start with a dedicated time slot to read the ADR. On average, 10 to 15 minutes should be enough. During this time, each team member reads the document and adds comments and questions to flag unclear topics. After the review phase, the ADR owner reads out and discusses each comment with the team.

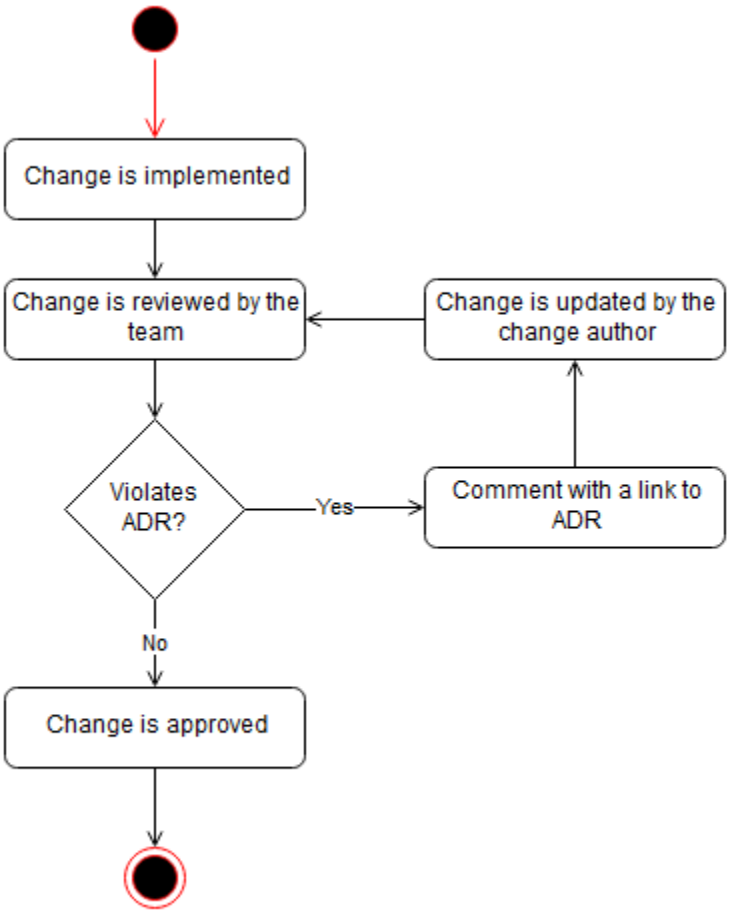
If the team finds action points to improve the ADR, the state of the ADR stays **Proposed**. The ADR owner formulates the actions, and, in collaboration with the team, adds an assignee to each action. Each team member can contribute and resolve the action points. It is the responsibility of the ADR owner to reschedule the review process.

The team can also decide to reject the ADR. In this case, the ADR owner adds a reason for the rejection to prevent future discussions on the same topic. The owner changes the ADR state to **Rejected**.

If the team approves the ADR, the owner adds a timestamp, version, and list of stakeholders. The owner then updates the state to **Accepted**.

ADRs and the decision log they create represent decisions made by the team and provide a history of all decisions. The team uses the ADRs as a reference during code and architectural reviews where possible. In addition to performing code reviews, design tasks, and implementation tasks, team members should consult ADRs for strategic decisions for the product.

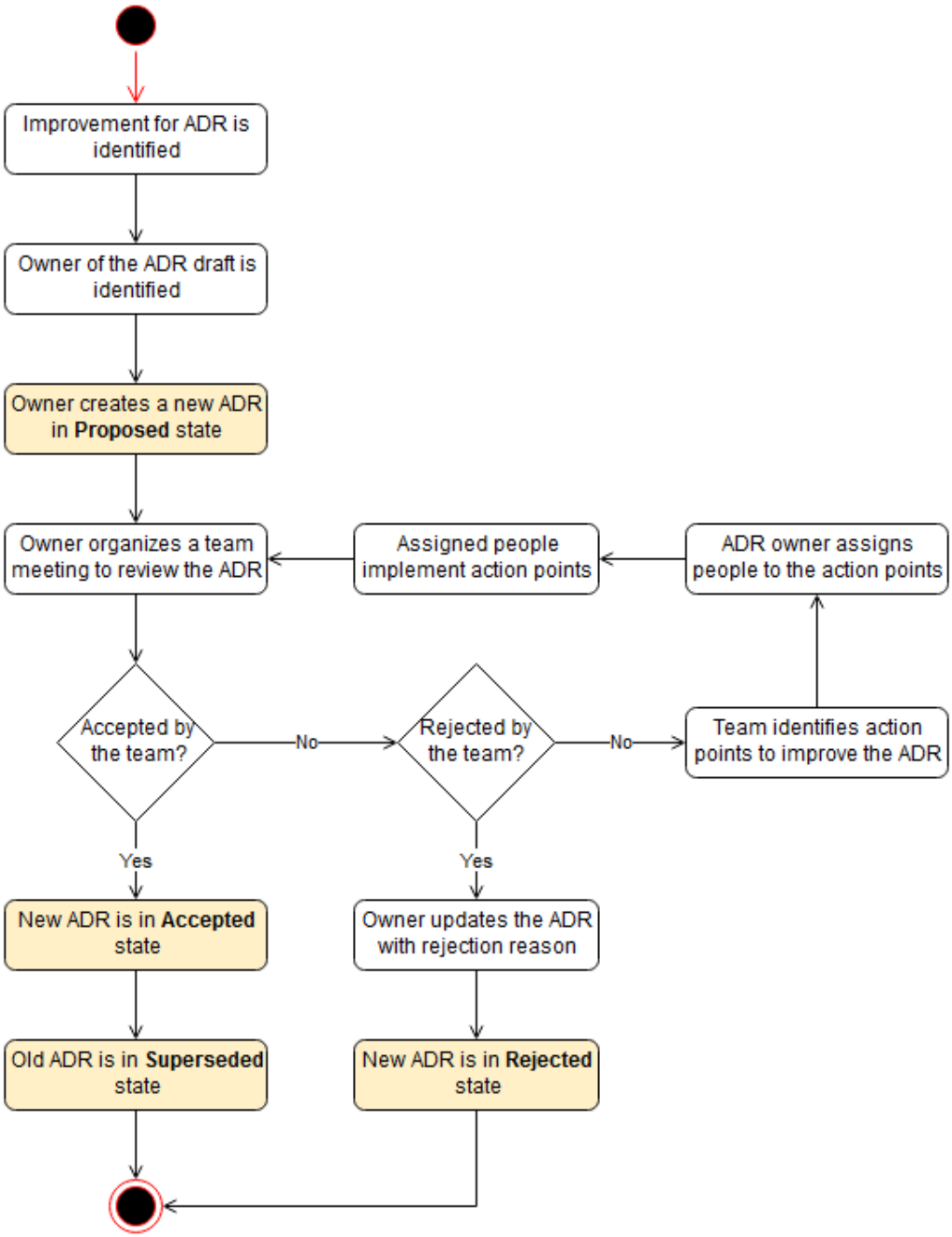
The following diagram shows the process of applying an ADR to validate if a change in a software component conforms to the agreed decisions.



As a good practice, each software change should go through peer reviews and require at least one approval. During the code review, a code reviewer might find changes that violate one or more ADRs. In this case, the reviewer asks the author of the code change to update the code, and shares a link to the ADR. When the author updates the code, it is approved by peer reviewers and merged into the main code base.

ADR review process

The team should treat ADRs as immutable documents after the team accepts or rejects them. Changes to an existing ADR requires creating a new ADR, establishing a review process for the new ADR, and approving the ADR. If the team approves the new ADR, the owner should change the state of the old ADR to **Superseded**. The following diagram illustrates the update process.



Best practices

Promote ownership. Each project team member should be empowered to create and own an ADR. This practice distributes architectural research work among team members and offloads that work from the solutions architect or team lead. It also fosters a sense of ownership in the decision-making process. This helps the team adopt those decisions faster instead of treating them as decisions that were imposed from higher levels of the organization.

Preserve ADR history. ADRs should have a change history, and each change should have an owner. When the ADR owner updates the ADR, they should change the status of the old ADR to **Superseded**, note their changes in the change history of the new ADR, and keep the old ADR in the decision log.

Schedule regular review meetings. If you are on a new (greenfield) project, the ADR process can be quite intense in the beginning. We recommend that you establish a cadence of regular ADR discussion and review meetings before or after the daily standup. With this approach, the defined ADRs will stabilize in two or three sprints, and you can build a solid foundation with fewer meetings.

Store ADRs in a central location. Each project member should have access to the collection of ADRs. We recommend that you store the ADRs in a central location and reference them on the main page of your project documentation. There are two popular options for storing ADRs:

- A Git repository, which makes it easier to version ADRs
- A wiki page, which makes the ADRs accessible to all team members

Address non-compliant code. The ADR process doesn't solve the issue of non-compliant legacy code. If you have legacy code that doesn't support the established ADRs, you can either update the outdated code base or artifacts gradually, while introducing new changes, or your team can decide to refactor the code explicitly by creating technical debt tasks.

FAQ

What are the benefits of creating an ADR process?

The project team should create an ADR process to streamline architectural decision-making, prevent repeated discussions about the same architectural topics, and communicate architectural decisions effectively.

When should the project team create an ADR?

The project team should create an ADR for every aspect of the software that affects structure (patterns such as microservices), non-functional requirements (security, high availability, and fault tolerance), dependencies (coupling of components), interfaces (APIs and published contracts), and construction techniques (libraries, frameworks, tools, and processes).

How often should the project team review an ADR?

The project team should review the ADR at least once before accepting it.

Who should create an ADR?

Every team member can create an ADR. We recommend that you promote a notion of ownership for ADRs. An author who owns the ADR should actively maintain and communicate the ADR content. Other team members can always contribute to an ADR. The ADR owner should approve changes to an ADR.

What information should an ADR contain?

At a minimum, each ADR has to define the context of the decision, the decision itself, and the consequences of the decision for the project and its deliverables. The context should mention possible solutions the team considered. It should also contain any relevant information relating to the project, customer, or technology stack. The decision must clearly state, in imperative language, the solution the team has decided to adopt. Avoid using words such as “should,” and phrase each decision to say “We use…” or “The team has to use…” The consequences section should mention all known trade-offs of making the decision. Each ADR must have a status and a changelog that contains the change date and the person who is responsible for the change.

Where can I find ADR templates?

There are multiple versions and variants of ADR templates available. For a public collection of commonly used ADR templates, see the ADR GitHub repository.

Next steps and resources

We recommend that you start small and see the benefit that ADRs bring to your team. If you are working on an ongoing project, identify the next architectural change and apply the proposed ADR process to create your first ADR.

Another starting point is to document your overall software development process by using ADRs. Often, the development process is based on tacit knowledge that the team didn't capture in any documentation. Documenting this process enables a smoother experience for new members of the team.

If you are on a greenfield project, apply the ADR process and start capturing all the decisions from the beginning in a few sentences. You can then iterate on those ADRs and supplement them with new information. After you establish your ADRs, you can start using them as a reference in your code review process.

Resources

- Architecture Decision Records. https://adr.github.io/.
- Richards, Mark and Neal Ford. 2020. *Fundamentals of Software Architecture*. Sebastopol: O'Reilly Media.

Appendix: Example ADR

Title

This decision defines the software development lifecycle approach for ABC application development.

Status

Accepted

Date

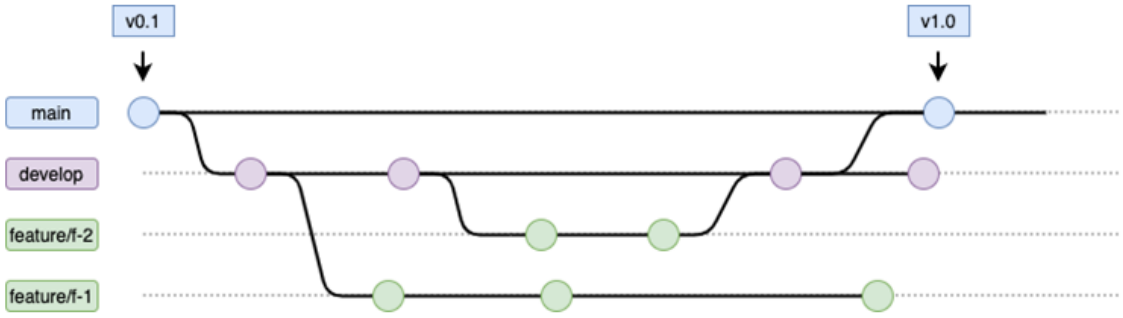
2022-03-11

Context

ABC application is a packaged solution, which will be deployed to the customer's environment by using a deployment package. We need to have a development process that will enable us to have a controllable feature, hotfix, and release pipeline.

Decision

We use an adapted version of the GitFlow workflow to develop ABC application.



For simplicity, we will not be using the `hotfix/*` and `release/*` branches, because ABC application will be packaged instead of being deployed to a specific environment. For this reason, there is no need for additional complexity that might prevent us from reacting quickly to fix bugs in production releases, or testing releases in a separate environment.

The following is the agreed branching strategy:

- Each repository must have a protected `main` branch that will be used to tag releases.
- Each repository must have a protected `develop` branch for all ongoing development work.

Consequences

Positive:

- Adapted GitFlow process will enable us to control release versioning of the ABC application.

Negative:

- GitFlow is more complicated than trunk-based development or GitHub flow and has more overhead.

Compliance

- The `main` and `develop` branches in each repository must be marked as `Protected`.
- Changes to the `main` and `develop` branches must be propagated by using merge requests.
- At least one approval is required for every merge request.

Notes

- Author: Jane Doe
- Version: 0.1
- Changelog:
  - 0.1: Initial proposed version

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication (p. 12)	—	March 16, 2022

AWS Prescriptive Guidance glossary

The following are commonly used terms in strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the **Provide feedback** link at the end of the glossary.

Modernization terms

business capability

What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

microservice

A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture

An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

modernization

Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment

An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)

Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

polyglot persistence

Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

split-and-seed model

A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

two-pizza team

A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development. For more information, see the Two-pizza team section of the Introduction to DevOps on AWS whitepaper.