Å
ENERGI

Guidelines for Software Development in Å Energi

Document responsible: DevEx team
Version: 1.0
Last changed: March 9, 2026

Å
ENERGI

Table of content:

Å
ENERGI

1 ABSTRACT

Å Energi's software guidelines align all teams around **security**, **quality**, **business value**, and **consistent engineering practices**. Every product team is accountable for its **end-to-end lifecycle**: Development, quality, security, operations, cost, and compliance. Developers are expected to collaborate, share knowledge, embrace change, and continuously improve.

The audience of this document is all new and existing developers in Å Energi Group. It should give a common foundation for how we develop software products that gives value for us and the society.

It should ensure that we deliver high-quality, sustainable, and secure software that enable Å Energi's strategic goals.

- Standardization & Consistency: To standardize practices across Architecture, DevOps and Technology, creating a unified development platform where all Product Teams operate predictably and consistently with our Cloud Native and Zero Trust principles.
- Enable Autonomy: To empower every Product Team with the guidance necessary to take full end-to-end ownership of their applications - including security, compliance, quality, and cost.
- Drive Efficiency: To maximize efficiency and foster collaboration across Å Energi by promoting automation, the use of Inner Source, and the re-use of shared components.

These guidelines cover architecture, DevOps, CI/CD, testing, infrastructure, feature toggling, code standards, AI tools, open/inner source and more.

2 CORE VALUES

All developers in Å Energi shall have the ambition to deliver optimal **business value** with **high quality** and **prioritize security** above all else. These guidelines will ensure a common understanding of principles and requirements that will steer us towards our common ambition.

- We share and build competence together
- We care about business value
- We communicate new ideas and opportunities
- We take security and quality seriously
- We embrace change and take ownership and responsibility

Å
ENERGI

3 HIGH LEVEL PRINCIPLES

1. *Product team (application responsible) are responsible for Compliance, Quality, Security, Cost, Operations and Development. The architecture and infrastructure should reflect this.*

2. *Shared infrastructure should only be used when strictly necessary. Cost, competence or complexity could be criteria's for sharing of infrastructure.*

3. *New applications should be based on cloud native principles in general.*

4. *The product team is responsible for control of integration with the outside world, both incoming and outgoing.*

5. *Product team shall constantly improve security and operations using automation and modern tooling.*

6. *Every product team shall expose security logs to Central Security Operation.*

7. *Every product team shall use Å Energi selected source control service for storing product code as well as code for configuration and infrastructure.*

8. *Every Product Team should participate in building a development community inside Å Energi Group with a shared and growing culture.*

4 ARCHITECTURE

4.1 Cloud Native Architecture

Cloud Native Architecture is an approach to building and deploying applications specifically designed for the cloud environment. It emphasizes certain key principles to maximize the benefits offered by cloud computing, leading to:

Scalability and Elasticity:
Applications are built with loosely coupled components that can be easily scaled up or down based on demand. This allows for efficient resource utilization and cost optimization. Application and infrastructure architecture should facilitate required scalability.

Resilience and Observability:
The architecture prioritizes fault tolerance and self-healing mechanisms to ensure continuous operation even if individual components fail. Systems are designed with built-in monitoring and logging capabilities to understand their behaviour and identify potential issues quickly. When errors do occur that the system cannot recover from, alerts should be raised to inform responsible parties.

Automation:
Repetitive tasks like provisioning, configuration and deployment are automated to minimize manual intervention and improve efficiency. This reduces human error and allows for faster development and deployment cycles. Using automated scripts that handle setting up new services, configuring software, and deploying updates, freeing up developers to focus on more creative tasks.

Containerization and Services:
Containerized applications are preferred as they can be built once and hosted in numerous ways. They are also bundled with all their dependencies on operating systems and installed frameworks making them portable across services and platforms. Azure has a wide range of services capable of hosting containers.

Applications are typically built as services that can communicate with each other using well-defined APIs. Use of monolith services, microservices or other solution architectures should be decided by the team to fit the case they are solving. Microservices can be developed, deployed and scaled independently, promoting agility and modularity. Monoliths can be less independently scalable and updateable, but can reduce complexity in terms of integrations, CI/CD and coupling compared to microservices.

Continuous Delivery:
Frequent releases and updates are encouraged through practices like continuous integration and continuous delivery (CI/CD). This enables faster feedback loops, quicker bug fixes and the ability to deliver new features and improvements rapidly.

By following these principles, cloud-native architecture empowers developers to build and deploy applications that are:

- Scalable: Adapt easily to changing demands.
- Resilient: Withstand failures and maintain continuous operation.
- Agile: Deliver new features and updates quickly.
- Cost-effective: Optimize resource utilization and minimize waste.

4.2 Legacy applications

Cloud-native applications should remain agile, loosely coupled and easy to evolve. When integrating with legacy systems, the goal is to avoid introducing sticky dependencies that slow down development, increase operational risk, or force you to inherit legacy constraints.

Principles for Integrating with Legacy Systems

- Treat legacy systems as external services, model legacy applications as external dependencies - even if they are internal to the organization. This keeps your domain clean and prevents tight coupling to old technology, data structures or release cadences.
- Prefer API-based integrations - use stable, versioned APIs or well-defined service contracts as the bridge between cloud-native apps and legacy systems
- Introduce an “anti-corruption layer” when needed. If the legacy application has messy data structures or inconsistent logic, add a small, isolated layer that transforms legacy models into clean domain models, adapts protocols and shields your services from legacy complexity
- Decouple through asynchronous messaging where possible, if the use case allows it, integrate through queues, event streams, or messaging patterns. This will reduce runtime coupling and give better scalability and resilience.
- Build for failure, legacy systems often have limited availability or unpredictable performance. Design your integration with, retry/backoff strategies, circuit breakers and fallbacks.

4.3 Clean Architecture

Clean architecture outlines a set of principles for designing software that prioritizes independence, maintainability, and testability. It achieves this through a layered approach and adherence to specific design principles like:

- Independent of Frameworks:
  The core business logic should not be dependent on specific frameworks or libraries. This allows for easier switching and avoids tying the application to specific technologies.
- Independent User Interface (UI):
  The core logic should be independent of the user interface, enabling the application to work with different UIs (web, mobile, etc.) without affecting the core functionality.
- Database Independent:
  The core shouldn't be concerned with the specific database technology used. Instead, it interacts with the database through an abstraction layer.
- Independent of External Elements:
  The core logic should be designed to function independently of any external dependencies, making it more adaptable and easier to test in isolation.
- Separation of Concerns:
  Clean architecture advocates for clear separation between different aspects of the application, such as the business logic, user interface, and data access. This promotes modularity and improves code maintainability.
- Dependency Inversion Principle:
  High-level modules (like the core) should not depend on low-level modules (like specific database implementations). Instead, both should depend on abstractions (like interfaces). This allows for easier testing and reduces coupling between components.
- Focus on Business Logic:
  The core of the application should focus solely on the core business logic and domain rules, free from any external concerns.

By adhering to these principles, clean architecture aims to create software that is:

- Maintainable:
  The clear separation of concerns and independent core make it easier to understand, modify, and extend the application over time.
- Testable:
  The core logic, being independent of external details, becomes easier to test in isolation, leading to more reliable and robust applications.
- Adaptable:
  The independence from specific frameworks and technologies allows the application to adapt to changes in the technological landscape more easily.

4.4 Azure Naming Conventions

Naming conventions are important for clean infrastructure code because they help us identify the purpose and function of different Azure resources. They also facilitate the management and maintenance of these resources, as well as the communication and collaboration among team members. Therefore, we follow a consistent and descriptive naming convention for our Azure resources that includes the following elements:

- Resource type: A prefix that indicates the type of the resource, such as vm for virtual machine, rg for resource group, or app for app service and so on.
- Environment: An optional suffix that indicates the environment of the resource, such as prd for production or dev for development.
- Product name: A short and unique name for the project that the resource belongs to, such as sales-reporting-solution, hr-hiring-portal or trading-platform.

For example, an app service that is used for the sales project in the production environment would be named app-prd-sales-reporting-frontend. A resource group that contains the app services for the trading project in the development environment would be named rg-dev-trading-platform. A naming convention like this helps us keep our code and Azure infrastructure clean and organized.
Our naming convention for Azure can be found here: Å Energi Azure Naming Convention

4.5 Zero Trust

Zero trust is a security model that assumes that any user, device or system, whether inside or outside the organization's network, could be compromised and should not be automatically trusted. Instead, every access request must be verified and authenticated before being granted. This approach helps to prevent unauthorized access and data breaches, and to protect the organization's assets and information.

For developers, adopting a zero-trust model means that they need to design and implement their software products with security in mind from the start. This includes using secure coding practices, such as input validation, error handling, and encryption, to prevent common vulnerabilities and attacks. Developers also need to integrate their products with the organization's identity and access management systems, to ensure that only authorized users can access the data and functionalities.

Additionally, developers should follow the principle of least privilege, granting users and systems only the minimum level of access necessary to perform their tasks.

4.6 FinOps

The team shall have processes to monitor and optimize cost.

- Know your costs, understand what drives cost in your solution and how services are billed

- Use cost tools like Azure Cost management to monitor and control, set cost alerts, budgets and dashboards.
- Evaluate if you can reduce cost with long term resources reservations
- Design for efficiency, avoid over-provisioning, use autoscaling and managed services and use storage lifecycle policies. Optimize compute, data transfer and caching.
- Perform cost reviews regularly

5 DEVOPS

In our company we adopt DevOps principles.
DevOps is a set of practices that combines software development and IT operations, aiming to shorten the development cycle and deliver software products faster and more reliably. DevOps emphasizes collaboration, automation, integration, and feedback across the entire software lifecycle, from planning to deployment and beyond. DevOps also promotes a culture of learning and experimentation, where failures are seen as opportunities for improvement and innovation. Some of the benefits of DevOps include increased customer satisfaction, reduced costs, improved quality and faster time to market.

DevOps practises:

- Version control system - Git, that allow the teams to manage and track changes to the source code and collaborate on the same code base.
- Continuous integration and continuous delivery (CI/CD) tools, that automate the building, testing, and deployment of the software products, ensuring that they are always ready for release. More on CI/CD in chapter 5.2
- Configuration management and infrastructure as code (IaC) tools, such as Bicep templates, that automate the provisioning and configuration of the software environments, ensuring that they are consistent and scalable.
- Monitoring and logging tools, such as Application insights or Prometheus, that provide visibility into the performance and health of the software products and the underlying infrastructure, enabling the teams to identify and resolve issues quickly.
- Collaboration and communication tools, such as Teams, that facilitate the coordination and information sharing among the teams and stakeholders, enhancing the culture of transparency and feedback.

By implementing DevOps, we aim to improve our agility, efficiency and quality, and to deliver value to our customers faster and more reliably.

5.1 Secure development lifecycle (SDLC)

A secure development lifecycle (SDLC) is a framework that guides the software development process and ensures that security is integrated into every phase. A SDLC typically consists of several stages, such as planning, analysis, design, implementation, testing, deployment, and maintenance. At each stage, security activities and controls are performed to identify and mitigate potential vulnerabilities and risks.

At our company, we follow a SDLC that aligns with the industry best practices and standards. We use various tools and methods to ensure the security of our software products, such as threat modelling, code review, static and dynamic analysis, penetration testing, and incident response. We also provide regular training and awareness programs to our developers and testers on the latest security trends and techniques.

By adopting a SDLC, we can achieve several benefits for our software development process and products. First, we can reduce the cost and complexity of fixing security issues in the later stages of development or after deployment. Second, we can improve the quality and reliability of our software products and meet the expectations and requirements of our customers and stakeholders. Third, we can enhance our reputation and trustworthiness as a software provider that values and respects the security and privacy of our users and data.

Å Energi has a Security Champion program with various training and workshops for developers. At least one from each product team should attend the program. The Security Champs are the main contact between the team and the security team. They have interest and build competence that improve the security in the team.

The following practices should be a part of any teams SDLC:

5.1.1 Design for security

Zero trust architecture and governance should be used in all stages in the software lifecycle. Protect the code from being changed by persons outside the team. Ensure that CI/CD process doesn't allow unauthorized users to alter the code. Ensure that your running environment follows best security practices across people, processes and technology. Monitor security vulnerabilities and incidents in all stages. Grant the least privilege required for each user account, machine/service identity and application component.

5.1.2 Security scanning in CI/CD

Use the CI/CD process to unveil vulnerabilities in your code and infrastructure before merge and deployment. In the pipelines we can do code analysis, secret scanning, dependency checks, container scanning, pen testing and more. Use the appropriate tooling that are already available in our platform.

5.1.3 Threat modelling and security review

Threat modelling is a structured approach to analysing the security design of the system, while "thinking like an attacker". The threats that are identified can then be mitigated before the new product or feature ships. Teams should do threat modelling at least once a year. We recommend the process threat modelling process described here - link.

Follow up of threats and tasks to mitigate them is included in ordinary work. The teams are responsible for prioritization and planning.

Compliance requirements assessment could be a part of the Threat modelling workshop, or a separate workshop build on the same methodology. Changes in compliance requirements could have impact of your system design.

5.1.4 Secure Infrastructure

The security of a workload relies on the security of the operational infrastructure and platform that is hosting it (cloud, on premises, hybrid, etc.). The security of the workload depends on the overall security of the environment including the security of identities, networks, servers, containers, email, user endpoints, and other systems. We use the Zero trust principles with least privilege required for each user account that have access. All users shall use multifactor authentication. For production environments we normally use privileged identity with eligible assignment.

Build artefacts running should be targeted by security scanning. A security scan should be capable of identifying vulnerabilities in operating systems, frameworks and packages. Azure Defender has these capabilities and should set up to monitor container registries, App Services and more.

When using containers, if your base image is present in the Å Energi internal Azure Container Registry, this should be used. This is to avoid problems with external dependencies such as Docker Hub and enables centralized security scanning of commonly used docker images.

5.2 CI/CD

One of the benefits of using CI/CD pipelines is that they enable fast and frequent feedback on the quality of the code and the functionality of the software. By automating the testing and deployment stages, we can reduce the risk of human errors and ensure that the code is always in a deployable state. This also allows us to deliver value to our customers faster and more reliably.
To leverage the full potential of CI/CD workflows, we need to follow some best practices, such as:

- Using code review tools and processes to ensure that the code is readable, consistent, and adheres to the team's quality criteria.
- Integrating code analysis tools and plugins into the pipeline to check for code smells, bugs, vulnerabilities, and other issues.
- Configuring the pipeline to run the unit tests and code analysis on every pull request and changes to main branch - reporting the results and status to the team.
- Using feature toggles to control the visibility and availability of new features and functionalities, enabling gradual and safe rollout and rollback. More on feature toggles in chapter 5.8
- Applying the principle of infrastructure as code (IaC), which means treating the infrastructure (such as servers, networks or databases) as code that can be written, tested, and deployed in a consistent and repeatable way. IaC helps us automate the provisioning and configuration of the infrastructure, reducing human errors, and increasing scalability and reliability.

Each product team is responsible for following best practices and creating and maintaining CI/CD pipelines for their products.

5.3 Automated testing

In general, code coverage is a measure of how much of the code is executed during testing. It is usually expressed as a percentage, with higher percentages indicating more thorough testing. While there is no universal standard for code coverage, many organizations aim for a certain level of coverage, such as 70% or 80%, to ensure that their code is adequately tested.

It is important to note that code coverage is just one aspect of testing, and high coverage does not necessarily guarantee high quality. Other factors, such as the quality of the tests themselves and the thoroughness of the testing process, also play a role in ensuring the quality of the software. Ultimately, it is up to each team to determine their own standards for code coverage and testing, based on their specific needs and goals.

We recommend teams to use automated testing, but we do not have any clear guidelines on code coverage etc. It is up to each team to own the quality of the products. Unit tests should run in our CI/CD workflows, and a test report should be generated and published. If tests fail the pipeline should fail and the code should not be deployed.
The team should aim to have a test suite they trust and when it reports success, the application should be ready for production.

5.4 Infrastructure as code

Infrastructure as Code (IaC) is a practice that involves automating the provisioning and configuration of software environments, ensuring that they are consistent and scalable. Bicep is a tool that can be used to implement IaC. It is a domain-specific language (DSL) for deploying Azure resources declaratively. It is designed to make it easier to write and maintain IaC templates for Azure Resource Manager (ARM). By using IaC and Bicep, you can automate the management of your infrastructure, reduce human errors, and increase scalability and reliability.

In Å Energy we create and store infrastructure files along with our source code in Git. We automate the infrastructure deployments with CI/CD. This allows us to quickly recreate or update infrastructure in Azure.

5.5 Packing Strategy

We use the following guidelines for creating and maintaining packages for our projects:

- We follow the standard conventions and best practices for each programming language and framework. For example, we use setup tools and requirements.txt or uv and pyproject.toml for Python, *npm and package.json for Node.js, and Nuget for dotnet.
- We use semantic versioning to label our packages with meaningful and consistent version numbers. We increment the major version when we introduce breaking changes, the minor version when we add new features, and the patch version when we fix bugs or improve performance.
- We document our packages thoroughly, including a README file that describes the purpose, features, installation, usage, and dependencies of the package, a LICENSE file that specifies the terms and conditions of using the package, and a CHANGELOG file that records the changes made in each version of the package.
- We test our packages before publishing them to ensure that they work as expected and do not introduce any errors or vulnerabilities. We use automated testing tools and frameworks fit for programming language and technical requirements.
- We use package repositories to publish packages we are going to distribute internally. We use private GitHub Package Registry or Azure Artifacts feed (deprecated) to store and manage our packages securely and reliably. We configure our projects to authenticate and access the feed as needed.
- We monitor and update our packages regularly to keep them compatible with the latest versions of their dependencies and the target environments. We use tools such as Dependabot and Renovate to automate the process of checking and updating our packages. We also review and address any issues or feedback from our users or customers.

5.6 Repository Strategy

Normally we use mono repos to avoid having a lot of repos with dependencies. A repository should represent a significant, cohesive part of your business logic (a "macro-service" or bounded context), not just a single microservice. If the mono repo grows too large you could split it with “separation of concerns” as a principle, ex infrastructure, frontend, backend etc.

Every repository should contain these specific files:

File	Purpose	Onboarding Value
README.md (required)	High-level overview, links to external docs and repository status badges.	**The Starting Point:** Tells the new user what the project is.
CONTRIBUTING.md	Details on branch naming, commit message conventions, and pull request review process.	**How to Work:** Teaches team workflow standards.
SETUP.md	**Step-by-step guide** on cloning the repo, installing dependencies, and running the service locally.	**The Quick Start Guide:** Gets a new user running code in minutes.

In addition, you should consider the need for LICENSE.md to define legal terms.

5.7 Environment strategy

For each system we have a clear strategy for environments that states the purpose of each running environment, and what role the environment plays in deliver secure and high-quality software.

Adopt short-lived environments instead of long-lived, both in the development phase and in the CI/CD process. This will reduce cost, reduce ops and improve security by reducing the attack surface. It will also improve consistency between the environments by using automated provisioning in the CI/CD.

In general, we should use synthetic test data in our test and development environments. Test or development environments that contains sensitive data is considered as production and shall be operated and secured as a production environment.

5.8 Feature toggling

A feature toggle is a technique that allows us to modify the behaviour of a system without changing the code. It involves adding conditional statements in the code that check the value of a toggle (usually a Boolean variable or a configuration parameter) and execute different branches of code depending on the value. For example, we can use a feature toggle to hide or show a new feature in the user interface, or to enable or disable an API endpoint.

- They enable faster and more frequent deployment, as we can deploy code that is not ready for release and keep it hidden until it is fully tested and validated.
- They allow us to experiment with new features and functionalities in production, and measure their impact and performance, without affecting the existing users or functionality.
- They provide flexibility and control over the release process, as we can activate or deactivate features at any time, without requiring code changes or redeployment.
- They reduce the risk and complexity of merging and branching, as we can avoid long-lived feature branches and merge the code to the main branch sooner, while keeping it isolated by the toggle.

However, feature toggles also have some drawbacks and challenges, such as:

- They increase the complexity and maintenance cost of the code, as we need to add and remove conditional statements and manage the toggle values across different environments and configurations.
- They introduce technical debt and code rot, as we need to clean up the unused code and toggles after the feature is released or discarded, and ensure that the code is consistent and coherent.

- They can cause unexpected interactions and conflicts between different features and toggles, especially if they are not well> documented and coordinated across the team and stakeholders.
- They can affect the performance and reliability of the system, as they add extra overhead and logic to the code execution, and may introduce bugs or errors if the toggle values are incorrect or inconsistent.

Therefore, it is important to follow some best practices when using feature toggles, such as:

- Use feature toggles sparingly and only for short-term experiments or releases, not for long-term or permanent changes.
- Define a clear and consistent naming convention and format for the toggles, and document their purpose, scope, and status.
- Use a centralized and externalized system to store and manage the toggle values and ensure that they are synchronized and accessible across different environments and components.
- Monitor and track the usage and impact of the toggles and collect feedback and metrics on the features and functionalities they control.
- Set a clear and realistic timeline and criteria for removing the toggles and the associated code and assign responsibilities and ownership for the cleanup process.

5.9 Clean Code

One of the principles of software engineering is to write clean code that is easy to read, understand, maintain, and reuse. Clean code is not only beneficial for the developers who write it, but also for the users who interact with it, the testers who verify it, and the managers who oversee it.

To achieve clean code, we follow coding standards for our main languages: Python, C#, and TypeScript. Coding standards are sets of rules and guidelines that define how to write and format code in a consistent and coherent way. They cover aspects such as indentation, spacing, comments, naming conventions, structure, and design patterns.

We also try to take the same courses on coding so that we are many people with the same perception of clean code. By learning from experts and best practices, we can improve our skills and knowledge and apply them to our projects. We also share our insights and feedback with each other and conduct regular code reviews to ensure the quality and readability of our code.
Another way we ensure clean code is by using tools such as linters and static code scanners to automatically quality check our code. These tools scan our code for issues such as syntax errors, logical errors, security risks, and bad practices, and suggest improvements or corrections.

All projects should contain editor config for formatting rules. Developers can then always run auto-format on their changes to ensure the code follows the established structure. Our goal is to have a unified and shared editor config, so all projects share the same rules.

6 USE OF AI IN DEVELOPMENT

We use AI tools to support our software development processes. These tools may be applied across all stages of the development lifecycle—from design and coding to testing and documentation.

6.1 Be mindful of what you share

You are responsible for ensuring that any data you provide to AI tools is appropriate for the context. Always understand what information you send to AI systems and what those systems generate in return.

6.2 You are accountable for the output

Anything produced with assistance from AI agents is ultimately your responsibility. This includes code, documentation, architectural suggestions, and any other generated content.
AI tools can produce incorrect, insecure, or unsuitable output—for example, flawed logic, insecure configurations, or code that exposes secrets. It is your job to maintain strong quality control.

6.3 Mandatory quality practices

AI does not replace established engineering routines. In fact, these become even more important:

- Code reviews
- Source criticism
- Security evaluation
- Functional and automated testing
- Dependency and license checks

Treat AI-generated content as if you wrote it yourself: review it thoroughly, validate assumptions, and ensure it meets our standards for quality, security, and maintainability.

6.4 Handling sensitive data

Be especially cautious with sensitive information, such as personal data. Such data must only be used for the original purpose for which it was collected.
It must not be repurposed or exposed to AI tools unless explicitly permitted.
For testing, experimentation, or development scenarios where real data is unnecessary or prohibited, always use synthetic or anonymized datasets. This minimizes the risk of data leaks and supports compliance with data protection regulations.

6.5 Recommended AI tools

For AI-assisted coding, the recommended and approved solution is the GitHub Copilot (paid license).

7 OPEN SOURCE

In Å Energi, we use open source software as a valuable resource to accelerate our development process and reduce costs. However, we also recognize the potential risks associated with using open source software, such as vulnerabilities and outdated libraries. To mitigate these risks, we have implemented several measures to ensure the security and reliability of our software products.

We use tools and services that automatically check for known vulnerabilities and alert us if any are found. We also regularly review and update our open source dependencies to ensure that we are using the latest and most secure versions. We do this as part of our automated builds and deployments, usually CI/CD pipelines.

Second, we use tools to manage and monitor our software and libraries. These tools alert us when we need to upgrade compromised software and libraries, helping us to keep our software up to date and secure. The SOC team will in some cases send out emails directly to the user to make sure they upgrade to latest version.

By following these practices, we can use open source software safely and effectively, while minimizing the risks and maximizing the benefits. It is very important to learn about open source licenses and the potential consequences by adopting source code from different projects that are under different licenses. Å Energi prefer using BSD-type of license, preferably the MIT license.

To manage diverse licenses effectively, dev teams should adopt an automated, policy-driven strategy that uses Software Composition Analysis (SCA) to enforce whitelists and maintain a Software Bill of Materials (SBOM) for full transparency and legal compliance across all dependencies.

8 INNER SOURCE

Inner sourcing is the practice of using open source software development principles and methodologies within an organization. It involves encouraging collaboration and code sharing among different teams, to improve software quality, reduce development time, and foster innovation. Inner sourcing can provide several benefits to an organization, including:

- Improved code reuse, reducing the need to reinvent the wheel.
- Increased collaboration and knowledge sharing, leading to better solutions and faster problem-solving.
- Enhanced transparency and visibility, allowing for better decision-making and risk management.
- Improved employee engagement and satisfaction, as they are empowered to contribute to projects outside their immediate team.

All source repositories on our preferred source control platform are open to internal users, to foster the imaginative mind who wants to explore the wild west of source code repositories within Å Energi. It is possible to request private repositories for particularly sensitive projects or compliance issues.

9 AGILE METHODOLOGY

Each team can implement or design their own methodology to fit their assignment. Most common are agile approaches with relatively small work increments that on their own should deliver business value.

A methodology should fit the nature of the assignment and the people working in the team. It should be designed to fit a purpose and continuously revised and updated to keep up with the problem it should solve.

We are currently using Scrum, Kanban and other methodologies in our teams.

10 DOCUMENTATION

Documentation is normally stored together with your source file. This will ensure that the documentation is aligned with the actual release of your system, and it is easily accessible for the developers.

The documentation should contain:

- High-level overview of your system
- How to get started as a new developer
- Details of the process for contributing
- Legal terms and conditions

- Run book for how to operate the system
- Decision record for deviation of the guidelines and common practice.

Preferred format is Markdown.

11 TECHNOLOGY

Å Energi has a corporate agreement with Microsoft as a provider of Cloud resources.

11.1 GitHub

At Å Energi, GitHub.com is the primary platform for code and collaboration. To ensure robust security, the organization leverages GitHub Advanced Security, which includes features such as code scanning (static code analysis), dependency review, and organization-wide vulnerability reporting. These tools automatically detect known vulnerabilities in the codebase and identify credentials that should not be part of the repository.

Dependabot is used to monitor dependencies and alert teams to security issues, while GitHub Copilot can assist in automatically fixing vulnerabilities. By integrating these security measures into the development workflow, Å Energi ensures that code is continuously monitored, risks are mitigated early, and only secure, high-quality software is delivered.

11.2 Azure Devops (deprecated)

While GitHub is the preferred place for code-collaboration, we still have solutions that reside in Azure Devops (AZDO). AZDO has most of the same capabilities as GitHub as it can integrate with GitHub for security scanning on repositories. Dependabot, Copilot assisted fixing and organization level overview is not available. Teams are encouraged to transition from AZDO to GitHub.

11.3 Azure

In Azure environments, robust security is achieved through a combination of proactive tools and best practices. Microsoft Defender for Cloud provides advanced threat protection for Azure resources, continuously assessing configurations and detecting vulnerabilities across virtual machines, databases, and containers. For containerized workloads, Defender includes specialized scanning capabilities that analyse container images for known vulnerabilities before deployment, ensuring that only secure images are used in production.

Additionally, developers are encouraged to follow secure coding practices, integrate with identity and access management systems, and apply the principle of least privilege. By leveraging Defender and automated container scanning, we can significantly reduce the risk of unauthorized access and data breaches, while maintaining compliance and operational excellence.
Application developers and maintainer have dashboards available to view and act on security score for their domains.

System to system authentication should be done using managed identity authentication wherever possible. In the cases where other authentication mechanisms are not possible, secrets used for authentication should be securely stored in secret stores such as Azure Key Vault. Read access to Key Vaults should be highly restricted. Other runtime settings for applications should be stored in configuration stores such as Azure Application Configuration.

11.4 Tech radar

We have a common Tech Radar shared in AZDO wiki (TechRadar - Overview). TechRadar represents the current state of which technologies should be used for various purposes. It's a guide for making technical decisions without having to verify them with technical stakeholders. It also allows us to experiment with new technologies safely by seeing what others have discovered, or not. The TechRadar shall also help us avoid tech spread.
Next iteration of the tech radar should include what technologies are in use, and what teams are using them. This will further our collaboration and knowledge sharing efforts.

11.5 Programming languages

Our main programming languages are C#, Python and Typescript:

- C# is a modern, object-oriented programming language developed by Microsoft, and is widely used for building Windows desktop applications and games.
- Python is a high-level, interpreted programming language known for its readability and versatility, and is commonly used for data analysis, and machine learning.
- TypeScript is a strict syntactical superset of JavaScript, developed and maintained by Microsoft, that adds optional type annotations and other features to the language, and is often used for web development.

For programming languages and their libraries, we aim to follow the vendors or community best practices in terms of code style and secure coding.