SETIC
Superintendência Estadual de
Tecnologia da Informação
e Comunicação

RONDÔNIA
Governo do Estado

Cartilha de Boas Práticas para Desenvolvimento de Sistemas SETIC

2026

GOVERNO DO ESTADO DE RONDÔNIA

Marcos José Rocha dos Santos
Governador

Sérgio Gonçalves da Silva
Vice-Governador

SUPERINTENDÊNCIA DE TECNOLOGIA DA INFORMAÇÃO E COMUNICAÇÃO

Delner Freire
Superintendente

Gabriel Carrijo Bento Teixeira
Diretor Técnico

COORDENADORIA DE DESENVOLVIMENTO

Janderson de Castro Thomaz
Coordenador

ELABORAÇÃO

Paulo Eduardo da Silva de Vasconcelos
Ádelle Camarão Monteiro
Matheus da Silva Cruz
Kleber Nogueira de Sá Júnior

REVISÃO

Paulo Eduardo da Silva de Vasconcelos
Ádelle Camarão Monteiro
Emilly Bezerra Miranda
Edilson Vasconcelos Dantas Junior
Gabriel Fernandes de Oliveira
Celso Dias de Oliveira Junior
Juliane Rodrigues Coutinho
Jônatas Justiniano Lima
Janderson de Castro Thomaz

PROJETO GRÁFICO

Idan Luiz Souza Santos

VERSÃO

VERSÃO	DATA	AUTOR	AÇÃO
1.0	30/12/2025	Equipe de Elaboração	Criação do documento

SUMÁRIO

1 INTRODUÇÃO 2
2 ORGANIZAÇÃO E PAPÉIS NO DESENVOLVIMENTO 3
2.1 PAPÉIS NO DESENVOLVIMENTO (SCRUM + *Quality Assurance*) 3
2.1.1 *Product Owner* (PO) 4
2.1.2 *Scrum Master* (SM) 4
2.1.3 Time de Desenvolvimento (*Dev Team*) 4
2.1.4 *Quality Assurance Tester* (QA) 4
2.1.5 Cliente e Usuário Final 4
2.1.6 Líder Técnico (*Tech Lead*) 4
3 ARQUITETURA DE SOFTWARE 6
4 DESENVOLVIMENTO DE CÓDIGO 7
4.1 CONVENÇÕES DE NOMENCLATURA – APLICAÇÃO (C#/.NET) 8
4.2 PADRÕES REST PARA APIs 9
4.2.1 Tabela – Verbos REST para API's 10
4.3 PADRÕES DE BANCO DE DADOS (DDL/DML) 12
4.4 PADRÕES DE *COMMIT*, *BRANCHES* E MENSAGENS 13
4.4.1 Fluxo de *Branches* (Git Flow adaptado) 13
4.4.2 Padrões de Mensagens de *Commit* (*Conventional Commits*) 14
4.5 *CHECKLIST* MÍNIMO DE *CODE REVIEW* 15
5 GESTÃO DE INCIDENTES E MUDANÇAS 19
6 TESTES E QUALIDADE 22
7 DOCUMENTAÇÃO DO SISTEMA 23
8 GOVERNANÇA E DIVULGAÇÃO 24
9 REFERÊNCIAS 26

1 INTRODUÇÃO

O desenvolvimento de sistemas no setor público e privado exige cada vez mais planejamento estruturado, padronização de processos e garantia de qualidade. Segundo Sommerville (2016), boas práticas em engenharia de software contribuem para reduzir falhas, aumentar a confiabilidade e otimizar a utilização de recursos, tornando o processo mais previsível e eficiente. Nesse sentido, a ausência de normas e orientações claras pode resultar em retrabalho, vulnerabilidades de segurança, baixa manutenibilidade e dificuldade de integração entre diferentes equipes.

Além disso, diretrizes institucionais como o Modelo de Referência MPS.BR e o CMMI (*Capability Maturity Model Integration*) reforçam a necessidade de se adotar processos documentados, padronizados e continuamente melhorados, mesmo em ambientes que não estejam formalmente buscando certificação. A aplicação de boas práticas permite alinhar os resultados da tecnologia da informação às estratégias organizacionais, garantindo maior valor agregado às soluções desenvolvidas.

Esta cartilha foi elaborada com o objetivo de fornecer orientações práticas e contextualizadas para as equipes de desenvolvimento de sistemas, servindo como guia para o planejamento, execução e manutenção de projetos. O documento contempla aspectos relacionados à organização de papéis, arquitetura de software, codificação, gestão de incidentes, testes, documentação e governança. Sua adoção visa fortalecer a cultura de qualidade, fomentar a colaboração entre equipes e assegurar que os sistemas desenvolvidos estejam alinhados às necessidades institucionais e sociais.

Portanto, mais do que um manual técnico, esta cartilha representa um instrumento estratégico de governança de TI, capaz de padronizar processos, minimizar riscos e promover a transparência no ciclo de vida do software.

2 ORGANIZAÇÃO E PAPÉIS NO DESENVOLVIMENTO

A ausência de definição clara de responsabilidades no desenvolvimento de sistemas pode gerar retrabalho, falhas de comunicação e atrasos. Nesse contexto, o Scrum destaca a importância da clareza de papéis para garantir transparência, inspeção e adaptação, permitindo que a equipe atue de forma colaborativa, autônoma e orientada à entrega de valor. Com base nisso, para uma visão mais detalhada pode ser consultado o Guia de Boas Práticas em Metodologias Ágeis, o qual se encontra por meio do link: https://wiki.setic.ro.gov.br/pt-br/home/gerencial/gerenciamentos/guia_de_agilidade.

Complementarmente, a utilização da Matriz RACI contribui para fortalecer a governança e reduzir ambiguidades, ao explicitar quem é responsável, quem aprova, quem deve ser consultado e quem deve ser informado em cada etapa. Assim, a organização de papéis deve integrar práticas ágeis e modelos de gestão de responsabilidades, assegurando eficiência operacional e alinhamento entre áreas técnicas e de negócio. Diante disso, a matriz elaborada pela Coordenadoria de Desenvolvimento da SETIC pode ser acessada através do link: https://wiki.setic.ro.gov.br/pt-br/home/spaces/code/gc/manuaiscode/matrizraci.

2.1 PAPÉIS NO DESENVOLVIMENTO (SCRUM + *Quality Assurance*)

No *Scrum*, os papéis fundamentais: *Product Owner*, *Scrum Master* e Time de Desenvolvimento são complementados pela prática institucional de incluir profissionais dedicados a testes para assegurar a qualidade (QA *Tester*), de forma a garantir maior robustez e imparcialidade na validação das entregas. A separação entre quem desenvolve e quem testa reforça a qualidade, evita viés e assegura que o software atenda aos requisitos funcionais e não funcionais.

2.1.1 *Product Owner* (PO)

Responsável por maximizar o valor do produto, definindo e priorizando o *backlog*. Atua como a voz do cliente e garante que o time trabalhe nas funcionalidades de maior impacto.

2.1.2 *Scrum Master* (SM)

Responsável por assegurar que o Scrum seja compreendido e aplicado corretamente. Atua como facilitador, removendo impedimentos e promovendo a melhoria contínua.

2.1.3 Time de Desenvolvimento (*Dev Team*)

Profissionais multidisciplinares que projetam, codificam e entregam incrementos de software prontos para produção. Devem seguir boas práticas de codificação, versionamento e revisão de código.

2.1.4 *Quality Assurance Tester* (QA)

Responsáveis por planejar, executar e automatizar testes (exceto testes unitários), assegurando que o software entregue atenda aos critérios de qualidade. Não testam o que desenvolveram, promovendo imparcialidade na validação.

2.1.5 Cliente e Usuário Final

Tanto o Cliente quanto o Usuário Final participam da homologação, fornecem feedback e validam se o software atende às necessidades reais de uso.

2.1.6 Líder Técnico (*Tech Lead*)

O Líder Técnico, que pode ser um desenvolvedor, atua como ponte entre a visão estratégica do produto e a execução técnica da equipe. Diferencia-se do *Scrum Master* (foco em processo) e do *Product Owner* (foco

em valor de negócio), pois sua responsabilidade principal é assegurar a excelência técnica, orientar boas práticas e apoiar a tomada de decisão arquitetural.

Boas Práticas

- Separação clara de funções: quem desenvolve não deve ser o mesmo que testa a funcionalidade, garantindo isenção.
- Testes colaborativos: embora o QA seja responsável por testes formais, os desenvolvedores devem realizar testes unitários, de integração e de regressão caso seja necessário antes da entrega.
- Definição de Pronto (Definition of Done): deve incluir critérios técnicos (código revisado, testes unitários, integração e de regressão caso seja necessário) e de qualidade (testes de sistemas executados por QA/DEV), pode ser consultado através do link: https://wiki.setic.ro.gov.br/pt-br/home/spaces/code/gc/padroes/developerguidelines/dod
- Integração contínua: automação de build e testes, permitindo feedback rápido sobre a qualidade do código.

3 ARQUITETURA DE SOFTWARE

A definição de uma arquitetura de software clara e padronizada é fundamental para assegurar escalabilidade, desempenho e manutenibilidade dos sistemas. Segundo Bass, Clements e Kazman (2013), a arquitetura de software fornece a estrutura fundamental do sistema, estabelecendo os componentes principais, suas responsabilidades e a forma como interagem entre si. A ausência de diretrizes arquiteturais pode levar a soluções fragmentadas, de difícil manutenção e pouco alinhadas às necessidades organizacionais.

No âmbito desta instituição, adota-se como padrão o Modelo MVC (*Model-View-Controller*) como Arquitetura Mínima Viável (AMV), amplamente reconhecido na literatura e na prática de mercado por separar de forma clara as responsabilidades entre dados (Model), interface de usuário (*View*) e lógica de controle (*Controller*). Essa abordagem favorece a modularidade, facilita a reutilização de código, simplifica a manutenção e permite maior flexibilidade na evolução dos sistemas (FOWLER, 2002).

Além disso, a utilização consistente de frameworks e bibliotecas alinhados ao MVC reduz riscos de inconsistência técnica, promove a padronização e contribui para a governança de TI, permitindo que equipes distintas desenvolvam soluções que compartilham a mesma base conceitual.

Boas práticas:

- Documentar a arquitetura mínima viável de cada sistema, destacando camadas, integrações externas, serviços e banco de dados.
- Adotar e padronizar o uso de *frameworks* que implementem o padrão MVC, assegurando consistência entre os projetos.
- Definir diretrizes para a integração de APIs e serviços, com foco em interoperabilidade e segurança.
- Aplicar práticas de versionamento (ex.: Git Flow) para garantir rastreabilidade e alinhamento da evolução arquitetural.

4 DESENVOLVIMENTO DE CÓDIGO

Códigos desenvolvidos sem padronização comprometem diretamente a manutenibilidade**,** legibilidade e evolução dos sistemas. De acordo com Pressman e Maxim (2016), a padronização no desenvolvimento de *software* é um fator crítico para reduzir complexidade, evitar ambiguidade e permitir que diferentes desenvolvedores compreendam e mantenham o código ao longo do tempo.

Nesse sentido, práticas como *Clean Code* (Martin, 2009) e os princípios SOLID fornecem diretrizes objetivas para produzir código claro, modular e de fácil manutenção. Além disso, a adoção de revisões de código (*code review*) contribui para a melhoria contínua da qualidade, possibilitando a identificação de defeitos precocemente e a disseminação do conhecimento entre a equipe (Rigby et al., 2013).

Outro ponto essencial é o uso sistemático de sistemas de versionamento, como Git, que asseguram rastreabilidade, controle de mudanças e colaboração efetiva entre desenvolvedores. Essa prática é considerada hoje indispensável em equipes que utilizam metodologias ágeis, como o Scrum, por apoiar a integração contínua e o desenvolvimento incremental.

Boas práticas:

- Estabelecer padrões de nomenclatura para variáveis, classes, métodos e APIs, promovendo consistência e legibilidade.
- Adotar os princípios de *Clean Code* (legibilidade, simplicidade e clareza) e SOLID (responsabilidade única, abertura/fechamento, substituição de Liskov, segregação de interfaces e inversão de dependência).
- Implementar revisões de código (*code review*) como etapa obrigatória antes da integração em *branches* principais.
- Registrar todas as alterações em sistemas de versionamento (ex.: Git), assegurando rastreabilidade e integração contínua.

- Promover boas práticas de documentação interna no código com a finalidade de evitar excessos e privilegiar a clareza do próprio código. Exemplo:

```
...
 // Aplica desconto promocional de aniversário: válido apenas no mês do cliente

decimal CalcularTotalComDescontoEspecial(decimal valor, decimal
percentualDesconto, DateTime dataNascimentoCliente)
{
    if (DateTime.Now.Month == dataNascimentoCliente.Month)
    {
        percentualDesconto += 0.1m; // bônus de 10% extra
    }
    decimal desconto = valor * percentualDesconto;
    return valor - desconto;
}
```

4.1 CONVENÇÕES DE NOMENCLATURA – APLICAÇÃO (C#/.NET)

- Projetos/*Namespaces*: Empresa.Produto.Modulo (ex.: Setic.Vendas.Api)
- Pastas (MVC): *Controllers*/, *Models*/, *Views*/, *Services*/, *Repositories*/, Dtos/, Mappings/
- Classes/Interfaces: PascalCase; interfaces prefixo I
- Métodos/Propriedades: PascalCase
- Variáveis locais: camelCase
- Constantes: UPPER_SNAKE_CASE
- Enums: PascalCase
- DTOs/ViewModels: sufixos Dto/ViewModel

Exemplo em C#:

```
public interface ICustomerRepository
 {
 Task<Customer?> GetByIdAsync(Guid id);
```

```
Task AddAsync(Customer entity);
}
public class CustomerService : ICustomerService
{
private readonly ICustomerRepository repository;

public CustomerService(ICustomerRepository repository)
{
this.repository = repository;
}
public async Task<CustomerDto?> GetAsync(Guid id)
{
var entity = await repository.GetByIdAsync(id);
return entity is null ? null : new CustomerDto(entity.Id, entity.Name,
entity.Email);
}
}
```

4.2 PADRÕES REST PARA APIs

API REST é uma interface baseada em recursos (*nouns*- “um dado ou entidade que pode ser manipulada por um cliente através da interface”) acessados via HTTP, que aplica princípios como *uniform interface*, *statelessness*, *cacheability* e *layered system*. Cada recurso tem um URI estável, é representado (geralmente) em JSON e manipulado por métodos HTTP padronizados, com códigos de status e semântica consistente (segurança, idempotência).

PADRÃO

Versão: versionar no caminho. (ex.: /api/v1/...)
Recursos no plural: /api/v1/customers, /api/v1/customers/{id}
Representação: Content-Type: application/json; charset=utf-8
Erro padronizado: RFC 7807 (application/problem+json)

- **type** → URI (link) que identifica o tipo de erro. Pode ser uma página de documentação.
- **title** → Título curto e padronizado do erro (ex.: "*Not Found*").
- **status** → Código de status HTTP (ex.: 404, 400, 500).

- **detail** → Mensagem explicativa em linguagem natural, para humanos entenderem o erro.
- **instance** → Indica a ocorrência específica do erro (ex.: qual *endpoint* ou recurso causou o problema).

```
{
  "type": "https://api.exemplo.com/erros/validacao",
  "title": "Dados inválidos",
  "status": 400,
  "detail": "Um ou mais campos estão incorretos.",
  "invalidParams": [
    { "field": "email", "reason": "Formato inválido" },
    { "field": "idade", "reason": "Deve ser maior que 18" }  ]}
```

Paginação: ?page=1&pageSize=50 (incluir X-Total-Count e Link quando possível)
Filtro/Ordenação/Seleção: ?status=active&sort=-createdAt&fields=id,name,status
Autenticação: Authorization: Bearer <token>
Traceabilidade: X-Request-Id em todas as respostas
Idioma: mensagens técnicas em inglês; mensagens ao usuário podem seguir o idioma institucional

4.2.1 Tabela – Verbos REST para API's

Verbo	Rota exemplo	Uso canônico	Corpo (req)	Status típicos	Seguro	Idempotente
GET	/customers, /customers/{id}	Ler recurso(s)	Não	200, 206, 304, 404	✓	✓
POST	/customers	Criar novo recurso	Sim	201,400, 409, 422	✕	✕
PUT	/customers/{id}	Substituição total	Sim	200/204, 400, 404, 409, 422	✕	✓

PATCH	/customers/{id}	Atualização parcial	Sim	200/204, 400, 404, 409, 422	✕	Depende
DELETE	/customers/{id}	Remover recurso	Não	204, 404	✕	✓
HEAD	/customers/{id}	Metadados sem corpo	Não	200, 404	✓	✓
OPTIONS	/customers	Capacidades e CORS	Não	204	✓	✓

Status = resposta que a API retorna.

Seguro = não altera nada.

Idempotente = repetir a operação não muda o resultado.

Corpo = se precisa ou não enviar dados junto a requisição.

Boas práticas (principal)

- Modelagem de recursos
  - Evite verbos na URL: prefira /invoices/{id}/payments a /payInvoice.
  - Use sub-recursos quando houver pertencimento claro.
  - Para ações não-CRUD, use POST em subcaminho de ação (/invoices/{id}:approve).]

- Códigos de status coerentes

  200 OK, 201 Created, 204 No Content, 400 Bad Request, 401/403, 404 Not Found, 409 Conflict, 412 Precondition Failed, 422 Unprocessable Entity, 429 Too Many Requests, 500/502/503.

- Validação e erro (RFC 7807)
  - { *"type": "https://api.example.com/errors/validation", "title": "Validation failed"* ... }

- Concurrency & Cache
  - *ETag/If-Match* para atualização condicional.
  - *ETag/If-None-Match + Cache-Control.*

- Paginação, ordenação e seleção
  - *?page=2&pageSize=20&sort=-createdAt&fields=id,name,createdAt.*

- Segurança
  - HTTPS obrigatório, *tokens Bearer*
  - *Least privilege* por escopos/*roles*; *rate limiting* + 429.

- Contratos e compatibilidade
  - Evitar *breaking changes*; usar versionamento.
  - Publicar *OpenAPI* atualizado.

- Operações assíncronas
  - Retornar 202 *Accepted* com *Location* /*jobs*/{*id*}.
  - O *job* expõe *status queued*, *running*, *succeeded* ou *failed*.

4.3 PADRÕES DE BANCO DE DADOS (DDL/DML)

A Coordenadoria de Desenvolvimento de Sistemas deve adotar integralmente os padrões de modelagem, nomenclatura e operação definidos pela Coordenadoria de Análise e Gestão de Dados – CAGD, garantindo uniformidade técnica, governança e integração entre sistemas. Esses padrões contemplam convenções para tabelas, colunas, chaves, índices, procedures e demais objetos DDL/DML, assegurando consistência e rastreabilidade em todo o ciclo de desenvolvimento.

Dessa forma, qualquer solução criada, evoluída ou mantida pela equipe de desenvolvimento deverá seguir as diretrizes estabelecidas pela área de Banco de Dados, evitando divergências estruturais, reduzindo riscos

operacionais e promovendo maior eficiência na manutenção e auditoria dos sistemas.

4.4 PADRÕES DE *COMMIT*, *BRANCHES* E MENSAGENS

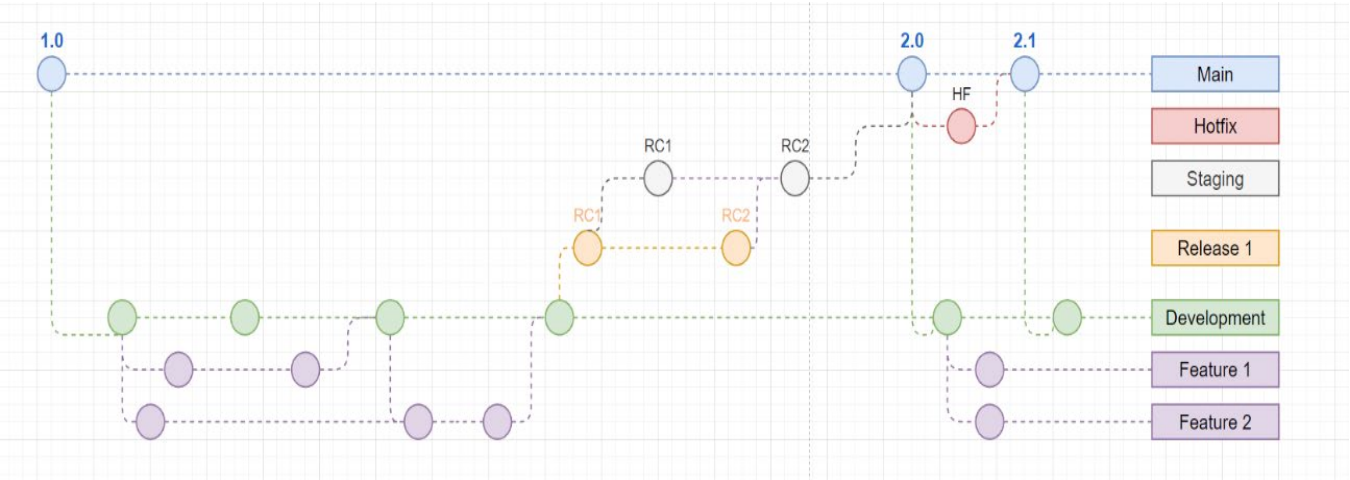
Um fluxo de versionamento bem definido é essencial para garantir organização, rastreabilidade e integração contínua no desenvolvimento de sistemas. Segundo Chacon & Straub (2014), o uso disciplinado de *branches* no Git possibilita paralelismo de desenvolvimento sem perda de controle. Além disso, mensagens de *commit* claras e padronizadas facilitam auditorias, revisões e manutenção do histórico do projeto.

4.4.1 Fluxo de *Branches* (Git Flow adaptado)

- *Main* (produção): contém apenas versões estáveis liberadas em produção.
- *Development* (desenvolvimento): integra novas funcionalidades e correções antes de irem para release.
- Feature *branches*: derivados de *development*, identificados como *feature*/<chave-*issue*> ou *feature*/<descrição>. Após concluídos e revisados, retornam para *development*.
- Release *branches*: derivados de *development*, usados para estabilização, testes e homologação antes do merge em *main*. Nomeados como *release*/<versão>.
- *Hotfix branches*: derivados de *main*, corrigem falhas críticas na produção. Nomeados como *hotfix*/<descrição>. Após corrigidos, devem ser mesclados em *main* e *development*.
- *Staging*: ambiente intermediário para testes integrados e simulações de produção antes da publicação definitiva.

- Representação gráfica: a imagem abaixo exemplifica como deve ser o fluxo de *branches*, incluindo versões, *releases*, *hotfixes* e *features*.

Figura – Representação Gráfica



Fonte – https://wiki.setic.ro.gov.br/home/spaces/code/gc/padroes/developer-guidelines/git

4.4.2 Padrões de Mensagens de *Commit* (*Conventional Commits*)

Mensagens de commit devem ser claras, objetivas e seguir o padrão:

<tipo>(escopo): descrição breve (#id_tarefa)

Exemplo:
feat(order): adiciona cálculo de frete (#123)

Tipos mais utilizados:

a. feat: nova funcionalidade
b. fix: correção de bug
c. docs: mudanças na documentação
d. style: ajustes de formatação
e. refactor: mudanças internas no código sem alterar comportamento externo
f. test: inclusão ou atualização de testes
g. perf: melhorias de performance

h. chore: tarefas de manutenção, configuração ou build
i. build: alterações que afetam o processo de build ou dependências externas

Boas Práticas

- *Commits* devem ser atômicos (resolver apenas uma tarefa).
  Exemplo:
  - *git commit -m "Corrige bug no cálculo de desconto em pedidos"*
  - *git commit -m "Ajusta cor e tamanho do botão 'Comprar'"*

Cada *commit* trata só uma coisa.

- *Commits* devem estar vinculados a *issues* ou tarefas no *backlog*.
- Não usar mensagens genéricas como “ajustes”, “teste” ou “alterações diversas”.
- O histórico deve contar a história do projeto de forma compreensível.

4.5 *CHECKLIST* MÍNIMO DE *CODE REVIEW*

O *Code Review* é uma prática essencial para assegurar qualidade, consistência e segurança no desenvolvimento de software. Mais do que identificar falhas, ele deve promover aprendizado coletivo e disseminação de boas práticas entre a equipe.

4.5.1 Objetivos da revisão de código:

- Identificar problemas de qualidade de código: A revisão de código ajuda a encontrar erros de sintaxe, bugs, problemas de lógica e outras questões que podem afetar a qualidade e funcionalidade do software;
- Garantir conformidade com padrões e boas práticas: A revisão de código assegura que o código siga as diretrizes e padrões definidos pela empresa, facilitando a manutenção e colaboração entre desenvolvedores.

- Melhorar compreensão e colaboração: Os revisores têm a oportunidade de entender melhor o código e oferecer feedback construtivo, promovendo o compartilhamento de conhecimento e a melhoria da equipe como um todo.
- Otimização e performance: A revisão de código permite identificar oportunidades para otimizar o código, melhorar o desempenho e reduzir a complexidade.

4.5.2 Itens a verificar:

A. Nomeação segue padrões estabelecidos?

- Variáveis, funções, classes e objetos de banco de dados seguem os padrões de nomenclatura definidos (camelCase, PascalCase, snake_case).
- A nomeação transmite claramente a responsabilidade do elemento, evitando abreviações obscuras e nomes genéricos.

B. Função/método segue o princípio da responsabilidade única (SRP/SOLID)?

- O método realiza apenas uma tarefa, com coesão e baixo acoplamento.
- Existe oportunidade de extrair lógicas complexas em funções auxiliares para melhorar clareza e reuso.

C. Tratamento de erros e logs estão consistentes?

- Erros são capturados adequadamente e tratados de forma a não comprometer a aplicação.
- Logs são informativos e não expõem dados sensíveis (ex.: senhas, *tokens*, dados pessoais).

- Níveis de log (INFO, WARN, ERROR) são utilizados corretamente.

D. Testes estão atualizados e executados no CI/CD?

- Foram criados ou atualizados testes unitários e de integração para cobrir as novas alterações.
- A cobertura de testes é adequada (não apenas o “caminho feliz”).
- O *pipeline* de integração contínua executa os testes automaticamente e eles passam sem falhas.

E. Consultas ao banco de dados estão otimizadas?

- Foram evitados problemas de performance como consultas N+1.
- Índices apropriados são utilizados quando necessário.
- As consultas utilizam SQL parametrizado, prevenindo injeção de SQL.
- Em listagens grandes há limites de paginação (LIMIT, TOP).

F. Segurança foi considerada?

- *Inputs* de usuário são validados e higienizados. (o que é higienizado?)
- Dados sensíveis são protegidos (*hash*/*salt* em senhas, criptografia) quando obrigatório, conforme LGPD.
- *Tokens*, chaves e credenciais não estão expostas no código.
- Políticas de autorização e/ou autenticação são aplicadas de forma consistente.

G. Documentação e artefatos relacionados estão atualizados?

- O código contém comentários claros quando necessário (sem excesso).

- O *README* do projeto reflete mudanças significativas (instalação, execução, variáveis de ambiente).
- *Scripts* de migração de banco de dados foram documentados e versionados.
- Existe referência a *issues*/tarefas no *backlog*, facilitando rastreabilidade.
- Nomeação segue padrões.
- Função/método faz só uma ação (SRP/SOLID).
- Tratamento de erros/*logs* consistente.
- Testes atualizados e passando na Integração Contínua
- Consultas otimizadas, como a não utilização de N+1.
- Segurança como SQL parametrizado, dados sensíveis protegidos etc.
- Documentação atualizada (comentários, README, migrações).

5 GESTÃO DE INCIDENTES E MUDANÇAS

A gestão inadequada de incidentes compromete a disponibilidade dos sistemas, a continuidade dos serviços e a satisfação dos usuários. Segundo a ITIL v4 (AXELOS, 2019), um incidente é qualquer evento que cause ou possa causar interrupção ou degradação da qualidade de um serviço. A falta de definição formal e de um fluxo estruturado para o tratamento de incidentes resulta em respostas lentas, inconsistentes e, muitas vezes, ineficazes.

Da mesma forma, a gestão de mudanças está diretamente ligada à confiabilidade e estabilidade do ambiente de TI. A ISO/IEC 20000-1 estabelece que todas as mudanças em serviços e sistemas devem ser avaliadas, autorizadas, priorizadas e registradas de forma a mitigar riscos e evitar impactos não planejados. Um processo bem definido de gestão de mudanças garante previsibilidade e maior controle sobre a evolução dos sistemas.

Dessa forma, é essencial adotar boas práticas que assegurem rastreabilidade, responsabilidade e transparência, permitindo identificar a causa raiz dos incidentes e garantir que as mudanças realizadas estejam alinhadas às necessidades institucionais.

Boas Práticas

- Definir formalmente o que caracteriza um *bug*, diferenciando-o de problemas, erros conhecidos e solicitações de serviço.
- Estabelecer um fluxo padronizado de tratamento, contemplando abertura, categorização, priorização, investigação, solução e fechamento.
- Garantir rastreabilidade por meio do registro de incidentes em sistemas de controle, vinculando-os a *commits*, versões ou mudanças aprovadas.
- Adotar critérios de classificação de severidade e níveis de prioridade, para assegurar respostas proporcionais ao impacto do incidente.
- Implementar uma gestão de mudanças estruturada, prevendo avaliação de impacto, autorização formal e documentação das alterações antes da implantação.

- Promover revisões pós-incidente (*post-mortem*), visando identificar causas raízes e implementar ações preventivas.

5.1 REGISTRO E ANOTAÇÃO DE BUGS E ERROS

O registro estruturado de bugs e erros é essencial para assegurar rastreabilidade, consistência e efetividade na identificação, análise e correção de falhas. Toda anomalia identificada deverá ser registrada no sistema oficial de gestão, contendo, obrigatoriamente, descrição clara e objetiva do problema, passos para reprodução, ambiente afetado, impacto identificado, evidências associadas (*prints*, *logs*, *payloads*) e critérios objetivos para validação da correção.

É imprescindível que cada bug seja devidamente categorizado e subcategorizado, de modo a permitir o mapeamento preciso dos tipos de falhas, a geração de estatísticas confiáveis e o monitoramento contínuo de tendências, recorrências e áreas críticas. Essa classificação estruturada favorece a priorização adequada, reduz ambiguidades, minimiza retrabalho e possibilita uma atuação integrada e alinhada entre as equipes de Desenvolvimento e Qualidade de Software (QA).

O histórico padronizado dos registros constitui insumo estratégico para análises de causa raiz, identificação de fragilidades sistêmicas e definição de ações preventivas e corretivas, contribuindo de forma consistente para a melhoria contínua da qualidade dos sistemas e dos processos de desenvolvimento.

Todos os procedimentos, critérios e padrões adotados para o registro, categorização e tratamento de bugs e erros deverão, obrigatoriamente, seguir o rito, as diretrizes e os padrões estabelecidos pela Gerência de Qualidade de Software (GQS).

Boas Práticas

- Implantar testes unitários, de integração, de sistemas e de aceitação, garantindo cobertura ampla e detecção precoce de falhas.

- Documentar casos de teste mínimos para cada funcionalidade, promovendo rastreabilidade e clareza nos critérios de validação.
- Definir critérios de aceite claros em conjunto com os stakeholders, assegurando alinhamento entre expectativas e entregas.
- Integrar os testes ao processo de Integração Contínua (CI), permitindo automação e execução frequente.
- Monitorar métricas de qualidade, como taxa de defeitos, cobertura de testes e tempo médio de resolução, para subsidiar a melhoria contínua.

6 TESTES E QUALIDADE

A ausência de testes sistemáticos e bem planejados resulta em falhas recorrentes em produção, retrabalho e custos elevados de manutenção. Segundo Myers, Sandler e Badgett (2011), o teste de software é um processo essencial para revelar defeitos, aumentar a confiabilidade e garantir que o produto atenda aos requisitos especificados. Além disso, a norma ISO/IEC 25010:2011 define qualidade de software como a capacidade do produto de satisfazer necessidades explícitas e implícitas, o que somente pode ser assegurado por meio de práticas consistentes de verificação e validação.

A literatura também destaca que os testes devem ser realizados em múltiplos níveis: unitários (foco em pequenos blocos de código), de integração (verificação da interação entre componentes) e de aceitação (validação junto ao usuário). Esses diferentes níveis permitem detectar falhas precocemente, reduzir riscos e assegurar maior robustez do sistema (Pressman & Maxim, 2016).

Assim, a qualidade não deve ser vista como uma etapa final, mas como uma prática contínua e integrada ao ciclo de desenvolvimento, em alinhamento com metodologias ágeis, como Scrum, onde a Definição de Pronto (*Definition of Done*) inclui a execução e aprovação dos testes.

Boas práticas

- Implantar testes unitários, de integração, de sistemas e de aceitação, garantindo cobertura ampla e detecção precoce de falhas.
- Documentar casos de teste mínimos para cada funcionalidade, promovendo rastreabilidade e clareza nos critérios de validação.
- Definir critérios de aceite claros em conjunto com os stakeholders, assegurando alinhamento entre expectativas e entregas.
- Integrar os testes ao processo de Integração Contínua (CI), permitindo automação e execução frequente.
- Monitorar métricas de qualidade, como taxa de defeitos, cobertura de testes e tempo médio de resolução, para subsidiar a melhoria contínua.

7 DOCUMENTAÇÃO DO SISTEMA

No desenvolvimento ágil, a documentação deve ser leve, objetiva e orientada a valor. O Manifesto Ágil (Beck et al., 2001) ressalta que *"software funcionando é mais importante que documentação abrangente"*. Isso, porém, não significa ausência de registros, mas sim a busca por uma documentação mínima viável, suficiente para garantir a continuidade, a manutenção e o alinhamento entre as equipes.

Conforme Sommerville (2019), a documentação bem estruturada permite que sistemas possam ser evoluídos e mantidos mesmo com mudanças na equipe, evitando a dependência de conhecimento tácito. Já a ISO/IEC/IEEE 26511:2018 reforça que os registros devem ser organizados e acessíveis, de modo a atender tanto aspectos técnicos quanto gerenciais.

Assim, a documentação deve equilibrar agilidade e governança, assegurando clareza e acessibilidade, sem burocratizar o fluxo de desenvolvimento.

Boas práticas:

- Produzir documentação mínima obrigatória, priorizando: visão geral do sistema, arquitetura, requisitos essenciais, fluxos principais e manuais de usuário.
- Centralizar os documentos em um repositório acessível, versionado e colaborativo (ex.: Wiki), integrando-o ao ciclo de desenvolvimento ágil.
- Atualizar a documentação de forma incremental e contínua, vinculada às sprints ou releases, garantindo que reflita o estado atual do sistema.
- Utilizar modelos padronizados e concisos, facilitando a escrita e leitura por diferentes equipes.
- Definir responsáveis pela curadoria da documentação, assegurando que a manutenção não fique restrita a indivíduos específicos.

8 GOVERNANÇA E DIVULGAÇÃO

Para que as boas práticas sejam efetivas, não basta apenas defini-las; é necessário que sejam institucionalizadas, divulgadas e incorporadas à cultura organizacional. A governança de TI, segundo o COBIT 2019 (ISACA, 2019), assegura que os investimentos em tecnologia gerem valor, com processos transparentes e alinhados às estratégias institucionais. Nesse sentido, a divulgação clara das diretrizes e a capacitação das equipes tornam-se pilares para o sucesso da implementação.

No contexto de metodologias ágeis, a governança não deve ser vista como algo burocrático ou restritivo, mas como um conjunto de mecanismos que apoia a autonomia das equipes e garante alinhamento institucional (Highsmith, 2010). A divulgação das boas práticas, quando feita de forma simples e acessível, promove transparência, engajamento e melhoria contínua.

Portanto, a cartilha deve ser tratada como um instrumento oficial de governança de TI, amplamente divulgado e revisado periodicamente, assegurando sua atualização e aderência às necessidades reais da organização.

Boas práticas

- Institucionalizar a cartilha como documento oficial, reconhecido pela gestão e integrado às normas internas de desenvolvimento de sistemas.
- Promover treinamentos, *workshops* e capacitações periódicas, reforçando a aplicação prática das diretrizes junto às equipes de desenvolvimento, teste e gestão.
- Publicar a cartilha em canais institucionais acessíveis (intranet, repositórios colaborativos, portais de TI), assegurando que todos tenham acesso às informações.
- Estabelecer um processo de revisão e atualização contínua**,** garantindo que as práticas evoluam conforme novas tecnologias, *frameworks* e necessidades organizacionais.

- Estimular uma cultura de aprendizado organizacional e melhoria contínua, na qual a cartilha sirva como referência viva, em constante evolução.

9 REFERÊNCIAS

AXELOS. *ITIL® Foundation: ITIL 4 Edition*. London: TSO (The Stationery Office), 2019.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software Architecture in Practice*. 3. ed. Boston: Addison-Wesley, 2013.

BECK, Kent et al. *Manifesto for Agile Software Development*. 2001. Disponível em: https://agilemanifesto.org/. Acesso em: 01 set. 2025.

FOWLER, Martin. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2002.

HIGHSMITH, Jim. *Agile Project Management: Creating Innovative Products*. 2. ed. Boston: Addison-Wesley, 2010.

ISACA. *COBIT 2019 Framework: Governance and Management Objectives*. Schaumburg, IL: ISACA, 2019.

ISO/IEC 20000-1:2018. *Information technology — Service management — Part 1: Service management system requirements*. Geneva: ISO, 2018.

ISO/IEC 25010:2011. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE)*. Geneva: ISO, 2011.

ISO/IEC/IEEE 26511:2018. *Systems and software engineering — Requirements for managers of user documentation*. Geneva: ISO/IEC/IEEE, 2018.

LAPKIN, Anne. *Best Practices for Incident and Problem Management*. Stamford: Gartner, 2018.

MARTIN, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River: Prentice Hall, 2009.

MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. *The Art of Software Testing*. 3. ed. Hoboken: Wiley, 2011.

PMI – Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*. 6. ed. Newtown Square: Project Management Institute, 2017.

PRESSMAN, Roger S.; MAXIM, Bruce R. *Engenharia de Software: uma abordagem profissional*. 8. ed. Porto Alegre: McGraw Hill, 2016.

RIGBY, Peter C. et al. Convergent Contemporary Software Peer Review Practices. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg: ACM, 2013.

SCHWABER, Ken; SUTHERLAND, Jeff. *The Scrum Guide*. 2020. Disponível em: https://scrumguides.org/. Acesso em: 01 set. 2025.

SOMMERVILLE, Ian. *Engenharia de Software*. 10. ed. São Paulo: Pearson, 2019.

WEILL, Peter; ROSS, Jeanne W. *IT Governance: How Top Performers Manage IT Decision Rights for Superior Results*. Boston: Harvard Business School Press, 2004.

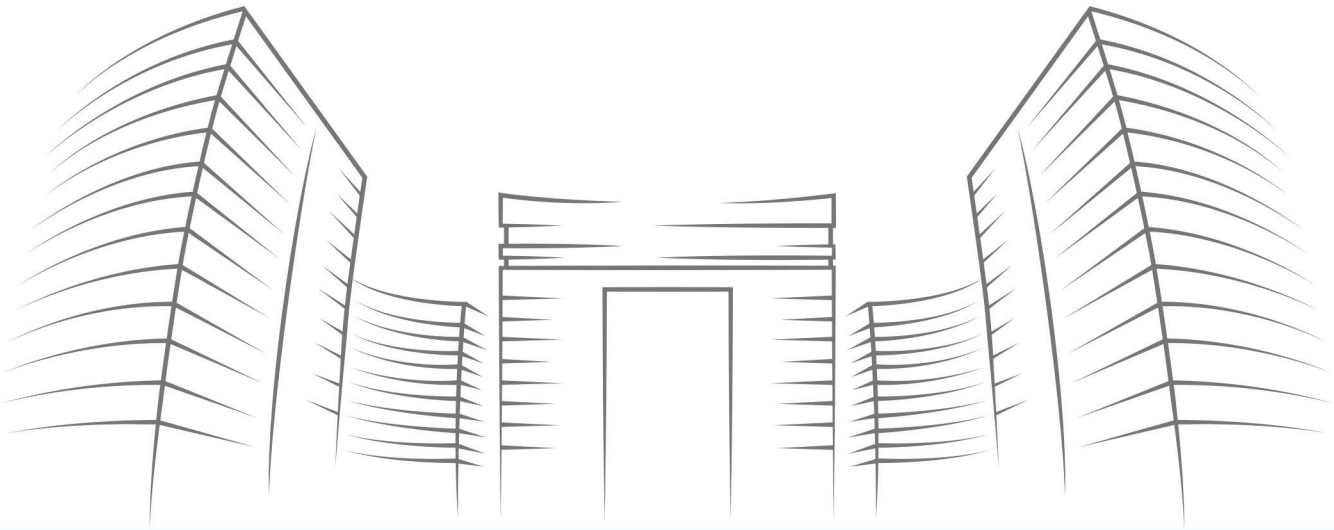
SETIC
Superintendência Estadual de
Tecnologia da Informação
e Comunicação

RONDÔNIA
Governo do Estado

Wiki.SETIC

Plataforma de Documentação Operacional e Gerencial dos Serviços da SETIC

wiki.setic.ro.gov.br



setic.ro.gov.br

@setic.rondonia

Telefone: 69 3212 9541
Endereço: Av. Farquar, 2986 - Bairro Pedrinhas
Palácio Rio Madeira, Edifício Rio Cautário - 6º Andar
Porto Velho, RO - CEP 76801470