ChatGPT

# Plano de Desenvolvimento da Disciplina Desenvolvimento de Aplicações para Dispositivos Móveis

## 1. Escopo de Aplicações Móveis Sugeridas

Para aplicar o desenvolvimento mobile cross-platform e boas práticas modernas, sugere-se escolher um projeto de aplicativo móvel **realista** que tenha um escopo significativo e abranja múltiplas funcionalidades. Abaixo estão algumas ideias de escopo de aplicação [1]:

- **Aplicativo de Delivery:** Uma plataforma estilo iFood/UberEats onde usuários podem encontrar restaurantes/lojas, fazer pedidos de entrega e acompanhar o status em tempo real. Inclui módulos de catálogo de produtos, carrinho/pedidos, pagamentos e rastreamento de entregadores. Esse escopo permite demonstrar o uso do Supabase para autenticação de usuários (clientes e entregadores), banco de dados de produtos/pedidos e sincronização em tempo real do status dos pedidos (por exemplo, atualizações ao sair para entrega).
- **Aplicativo de Finanças Pessoais:** Uma aplicação para controle de gastos e gerenciamento financeiro pessoal. Permite ao usuário registrar despesas e receitas, categorizar transações, visualizar gráficos/resumos mensais e definir orçamentos. Poderia incluir múltiplos perfis (caso família) e sincronização na nuvem dos dados financeiros. O Supabase se encaixa bem aqui armazenando as transações em um banco relacional seguro e fornecendo autenticação e armazenamento de recibos/imagens, enquanto o Flutter exibiria gráficos (podendo usar bibliotecas de chart) de forma elegante.
- **Aplicativo de Rede Social Local:** Uma rede social de alcance local/comunitário, onde usuários em uma determinada região podem postar conteúdo, comentar e interagir. Teria um feed de posts, sistema de amigos/seguidores, chat simples e notificações. Esse escopo explora recursos em tempo real (atualização instantânea do feed ou chats via Supabase Realtime) e upload de mídia (fotos/vídeos curtos) usando o Storage do Supabase, além de autenticação de usuários e regras de acesso a dados (p. ex. posts visíveis apenas a certos grupos).

Cada uma dessas ideias de projeto possibilita aplicar uma arquitetura mobile moderna e tirar proveito do Flutter com Supabase. Por exemplo, no caso do app de delivery, poderíamos ter funcionalidades de acompanhamento de entregas ao vivo e divisão de papéis (cliente, entregador, admin), demonstrando uso de autenticação robusta e atualizações em tempo real de dados [2]. Para fins deste plano, assumiremos a primeira opção (aplicativo de delivery) como exemplo base para detalhar arquitetura e tecnologias, mas o planejamento geral serve a qualquer dos escopos sugeridos.

## 2. Arquitetura Proposta da Aplicação

A arquitetura geral proposta segue um modelo cliente/servidor simplificado: o aplicativo Flutter (cliente) se comunica diretamente com um backend *serverless* baseado em **Supabase**, que provê banco de dados Postgres, autenticação, APIs instantâneas e outros serviços em nuvem [3]. Em resumo, a pilha será bem objetiva – Flutter no front-end e Supabase no back-end [4] – aproveitando ao máximo a

infraestrutura pronta do Supabase para evitar a necessidade de criar servidores dedicados. Os principais componentes dessa arquitetura são:

- **Aplicativo Mobile (Flutter):** O app Flutter será responsável pela interface com o usuário e lógica de apresentação. Sendo um app *cross-platform*, o mesmo código atenderá Android e iOS. Ele consumirá os serviços do Supabase via SDK (por exemplo, usando o pacote `supabase_flutter`) ou requisições REST, conforme o caso. O app gerencia a navegação de telas, captura de entradas do usuário e exibição de dados. Deve ser estruturado de forma a separar a camada de UI da lógica de negócio (ver seção de boas práticas), usando gerência de estado adequada para refletir os dados obtidos do back-end. O Flutter oferece recurso de *hot reload* para agilizar o desenvolvimento e widgets nativos (Material Design para Android, Cupertino para iOS) para uma UI consistente.

- **Backend como Serviço (BaaS) – Supabase:** A aplicação utilizará o Supabase como back-end, evitando a necessidade de desenvolver servidores do zero. O Supabase fornece uma série de ferramentas integradas sobre um banco **Postgres** na nuvem [2]. Cada projeto Supabase já inclui:

- **Banco de Dados Postgres (API REST):** O banco de dados relacional armazena todas as informações da aplicação (por ex.: usuários, pedidos, produtos, posts, transações financeiras, etc.). O Supabase expõe automaticamente uma API RESTful para as tabelas (via PostgREST) permitindo ao app executar operações CRUD simplesmente chamando endpoints ou usando o SDK. Isso significa que não é necessário implementar uma API manualmente – as tabelas e *views* no Postgres já ficam acessíveis de forma segura. Opcionalmente, pode-se habilitar o Supabase para fornecer APIs **GraphQL** se desejado. Cada tabela terá políticas de segurança (RLS) definindo quais usuários podem ler/escrever cada dado, garantindo isolamento conforme a lógica de negócio.
- **Autenticação de Usuários:** O Supabase Auth gerencia o cadastro, login e autenticação dos usuários, oferecendo suporte a email/senha e autenticação social (OAuth) out-of-the-box. Os usuários autenticados recebem *JSON Web Tokens* (JWT) para acessar os serviços, e o Supabase aplica **Row Level Security** no banco com base nesses tokens, garantindo que cada usuário só acesse seus dados [5]. Por exemplo, no app de delivery, entregadores e clientes podem ter papéis distintos controlados via claims no JWT ou colunas de perfil. A integração de Auth no Flutter é facilitada pelo SDK (`supabase.auth`) que fornece métodos para login, logout, reset de senha etc.
- **Sincronização em Tempo Real (Realtime):** Um dos pontos fortes do Supabase é o suporte a dados em tempo real. Através do recurso *Realtime*, o aplicativo pode **assinar** (*subscribe*) atualizações em tabelas ou canais e receber eventos via WebSocket sempre que os dados mudarem [6]. Isso possibilita implementar funcionalidades como: atualização instantânea do status de um pedido de delivery na tela do cliente, feed de posts em tempo real na rede social local, ou ainda presença online/indicador de digitando em um chat. No Supabase, o realtime é implementado sobre *Postgres Logical Replication*: ao ativar a replicação em uma tabela, quaisquer inserts/updates/deletes nela geram eventos para os clientes conectados. O Flutter, usando o SDK (`supabase.channel`), pode escutar esses eventos e assim atualizar a interface automaticamente, mantendo os dados sincronizados sem que o usuário precise refrescar a tela.
- **Armazenamento de Arquivos (Storage):** Para conteúdos binários (fotos, vídeos, arquivos em geral) que o app precisar salvar, o Supabase oferece um serviço de storage S3-like. Por exemplo, imagens de perfil, fotos de produtos ou comprovantes podem ser armazenados com segurança e referenciados no banco de dados. O Flutter pode enviar e baixar arquivos usando o SDK do Supabase, e as regras de acesso (políticas) garantem que cada usuário só acesse seus arquivos

autorizados. Esse serviço abstrai a necessidade de configurar um bucket AWS S3 manualmente – tudo é gerenciado dentro do projeto Supabase.

- **Funções *Edge* (Serverless):** Caso seja necessária lógica de negócio no backend que vá além do CRUD básico (por exemplo, processamento de pagamentos com terceiros, envio de notificações por e-mail/SMS, ou lógica pesada que não se queira fazer no app), o Supabase disponibiliza funções serverless (Edge Functions) escritas em TypeScript/JavaScript (runtimes V8/Deno). Essas funções podem ser desencadeadas por eventos do banco ou chamadas via HTTP pelo app. No contexto do nosso projeto, poderíamos usar Edge Functions para, por exemplo, enviar uma notificação push via Firebase Cloud Messaging quando um pedido muda de status, ou integrar com uma API externa (como Google Maps para calcular rotas de entrega). Embora não obrigatório, este componente está disponível para extensibilidade futura sem precisar de servidor próprio.

- **Serviços Externos e Integrações:** Além do Supabase, a arquitetura pode incluir integrações com serviços de terceiros conforme as necessidades do projeto. Por exemplo, para notificações *push* em mobile (algo não fornecido nativamente pelo Supabase), pode-se integrar o Firebase Cloud Messaging (FCM) ou OneSignal; para mapas e geolocalização, usar a API do Google Maps via plugin do Flutter; para processamento de pagamentos, usar SDKs de pagamento (PagSeguro, Stripe) diretamente no app ou via funções serverless. Esses componentes externos funcionariam de forma complementar: o app Flutter chamaria os SDKs apropriados ou endpoints das funções edge. É importante gerenciar de forma segura as chaves de API desses serviços (usando armazenamento seguro ou variáveis de ambiente no app, quando possível). Em suma, a arquitetura permanece modular: o Flutter lida com a experiência do usuário e integra plugins conforme necessário, enquanto o Supabase centraliza os dados e regras de negócio principais no backend.

Essa arquitetura proposta aproveita ao máximo a **estrutura pronta do Supabase**, reduzindo a configuração de infraestrutura para os alunos, e enfatiza a divisão clara entre front-end e back-end. O app Flutter atua como cliente apresentacional, enquanto o Supabase assume responsabilidades tradicionais de servidor (persistência, autenticação, regras de acesso, backend em tempo real) [2]. Isso permite que os alunos foquem em implementar funcionalidades e experiência de usuário, ao mesmo tempo em que aprendem conceitos de backend (estruturas de dados, segurança, APIs) de forma prática e integrada.

## 3. Tecnologias Recomendadas

Para implementar a arquitetura e funcionalidades acima, são recomendadas tecnologias e ferramentas modernas alinhadas com o ecossistema Flutter e as exigências do projeto:

- **Linguagem e Framework de UI:** *Dart* com o framework **Flutter** (na sua versão estável mais recente). O Flutter possibilita desenvolvimento multiplataforma nativo para Android e iOS a partir de uma única base de código. Os alunos já familiarizados com conceitos de POO em C# e lógica em C rapidamente se adaptam ao Dart, que é fortemente tipado e suporta programação reativa. O Flutter traz uma rica biblioteca de widgets reutilizáveis seguindo **Material Design 3** (padrão visual do Android) [7], além de componentes *Cupertino* nativos do iOS, garantindo **UI/UX** consistente em ambas plataformas. Recomenda-se seguir as diretrizes oficiais de design do Material You e Human Interface Guidelines (iOS) ao criar as telas. Para acelerar o design de interfaces, pode-se aproveitar pacotes da comunidade: por exemplo, **GetWidget**, uma biblioteca open-source com mais de 1000 componentes de UI pré-construídos, que pode agilizar o desenvolvimento e é altamente customizável [8]. Outra possibilidade é o uso das bibliotecas

**Syncfusion Flutter** (oferece gráficos, calendários e outros widgets avançados úteis, por exemplo, gráficos para o app de finanças) sob licença comunitária gratuita. Contudo, mesmo sem bibliotecas extras, o kit padrão do Flutter é suficiente para a maior parte das necessidades, devendo-se introduzir pacotes externos apenas para funcionalidades específicas (como gráficos, mascaramento de campo, etc.) conforme o escopo demandar.

- **Backend como Serviço: Supabase** será a plataforma de back-end. O Supabase se autodenomina *The Open Source Firebase Alternative*, fornecendo um conjunto completo de funcionalidades de back-end a partir de um banco de dados Postgres [3]. Dentro do Supabase, serão utilizadas as seguintes tecnologias integradas:

- *Database:* PostgreSQL 14+ (com extensões como PostGIS se necessário para geodados, e pgVector se algum recurso de IA/vetores for explorado). O Postgres dá robustez, consistência e suporte a SQL, facilitando consultas complexas caso precisem ser feitas.
- *Auth:* Serviço de autenticação do Supabase para registro/login de usuários, com suporte a OAuth (Google, GitHub, etc.) se desejado. O uso do Supabase Auth simplifica a gestão de usuários e aplica políticas de segurança (RLS) diretamente nas tabelas do Postgres [5].
- *Storage:* Armazenamento de objetos para salvar imagens e arquivos do aplicativo, com interface HTTP e controle de permissões integrado.
- *Edge Functions:* Como mencionado, para lógicas customizadas. Podem ser escritas em JavaScript/TypeScript (Deno) e disparadas via eventos ou HTTP.

O Supabase possui um SDK oficial para Dart/Flutter (`supabase_flutter`) que deverá ser utilizado para interagir com esses serviços de forma segura e fácil. Com ele, é possível, por exemplo, fazer autenticação, executar consultas e *inserts* no Postgres, escutar eventos em tempo real e fazer upload/download de arquivos, tudo via código Dart. Isso elimina a necessidade de o aluno realizar requisições HTTP de baixo nível ou implementar clientes WebSocket manualmente – o SDK abstrai essas operações. **Observação:** Será necessário configurar no Supabase as políticas de acesso (RLS) para assegurar que as regras de negócio (quem pode ler/escrever cada dado) sejam respeitadas. Durante o desenvolvimento, é possível utilizar a chave *anon public* fornecida pelo Supabase (com RLS ativo) no app Flutter, e a chave de serviço (mais poderosa) apenas em ambiente seguro (por exemplo, nas Edge Functions, se usadas).

- **IDE e Ferramentas de Desenvolvimento:** Recomenda-se o uso de uma IDE completa para o desenvolvimento Flutter. As opções mais comuns são **Android Studio** (ou IntelliJ IDEA) com o plugin Flutter, ou **Visual Studio Code** com extensões Flutter e Dart. Essas IDEs oferecem autocompletar, debugger embutido, e facilitam o gerenciamento de pacotes. Para depuração visual e análise de performance do aplicativo, a ferramenta **Flutter DevTools** é essencial – ela permite inspecionar a árvore de widgets em tempo real, monitorar uso de memória, FPS, rastrear *layout* e *repaints*, ajudando a otimizar a fluidez do app. A **Hot Reload** do Flutter será muito utilizada para acelerar o teste de mudanças de UI durante as aulas/labs. Além disso, ferramentas de emulação e teste em dispositivo real: o Android Studio inclui um emulador de Android; é recomendável testar em pelo menos um dispositivo físico Android e, se possível, em um iPhone (ou usar o simulador iOS num Mac) para garantir compatibilidade.

- **Gerenciamento de Estado:** Para evitar código desorganizado e *bugs* provenientes de estado inconsistentes, é recomendada a adoção de um **gerenciador de estado** maduro no Flutter. Duas opções populares são o **Riverpod** e o **BLoC** (Business Logic Component). O **Riverpod** é uma biblioteca inspirada no Provider, porém mais robusta e com segurança em tempo de compilação, que facilita a injeção de dependências e gestão reativa do estado sem depender do `BuildContext` do Flutter. Já o **Flutter Bloc** implementa o padrão BLoC (amplamente usado em Flutter), separando nitidamente a lógica de negócio (em *blocs/cubits*) da interface, através de

fluxos de eventos e estados. Ambas abordagens são válidas – a escolha pode ficar a critério da equipe, mas o importante é **utilizar alguma forma de gerenciamento de estado** ao invés de depender apenas de variáveis globais ou `setState` espalhados. Isso garantirá um código mais previsível, modular e testável. (Observação: Em projetos menores, o próprio `setState` e callbacks podem ser suficientes, mas dado que o app proposto tende a crescer em escopo, é preferível desde cedo estruturar a lógica com um padrão de estado adequado). Caso a opção seja por Riverpod, pode-se usar os `StateNotifier` / `StateNotifierProvider` para estados mais complexos ou `FutureProvider/StreamProvider` para integrar com dados assíncronos (ex.: resultados do Supabase). No caso do BLoC, seguir a arquitetura com *Repositories* e *Use Cases* pode ajudar a isolar acessos ao Supabase (ex.: um repositório faz as chamadas Supabase e o BLoC apenas solicita ao repositório e emite estados para a UI).

- **Bibliotecas e Pacotes Auxiliares:** Além das já citadas (supabase_flutter, Riverpod/BLoC), diversos pacotes do pub.dev podem ser utilizados para agregar funcionalidades ou facilitar o desenvolvimento. Alguns exemplos: o pacote **http** ou **Dio** (cliente HTTP, embora para Supabase em particular o SDK cubra quase tudo via WebSockets/HTTP próprio), **intl** (formatação de datas, moedas – útil no app de finanças), **provider** (se optasse por Provider para estado), **firebase_messaging** (caso implementem notificações push via FCM), **geolocator** e **google_maps_flutter** (no app de delivery, para localização e mapas), **charts_flutter** ou **fl_chart** (gráficos no app de finanças), **image_picker** (upload de imagens da galeria/câmera, útil para fotos de perfil ou comprovantes). A seleção exata deve ser feita conforme as funcionalidades do projeto escolhido, evitando sobrecarregar o app com dependências desnecessárias. Sempre verificar a reputação e manutenção do pacote antes de adotá-lo.

- **Ferramentas de Teste e Qualidade:** Para garantir a qualidade do software, aproveitaremos as ferramentas de teste do próprio Flutter/Dart. O **Flutter Test** framework permite escrever testes de unidade (para funções Dart puras, como verificações de regras de negócio ou manipulação de dados) e **testes de widgets** (para verificar que componentes da UI se comportam conforme esperado em isolamento). Por exemplo, pode-se testar que um widget de formulário valida corretamente a entrada do usuário. Além disso, o Flutter oferece a biblioteca **integration_test** para testes de integração ponta-a-ponta, nos quais um código de teste controla o app rodando em um dispositivo/emulador, simulando taps e textos como um usuário faria. Esses testes E2E podem validar fluxos completos (ex.: cadastro e login de usuário, realizar um pedido e ver o status). É recomendável configurar pelo menos alguns testes automatizados para as partes críticas da aplicação. Para facilitar a escrita de testes, bibliotecas como **Mockito** podem ser usadas para criar *stubs* dos serviços (por ex., simular respostas do Supabase em testes unitários da lógica, isolando do backend real). Ferramentas de análise estática como **dart analyze** (ou a integração de linters no VSCode/Android Studio) ajudarão a manter o código aderente às boas práticas do Dart. Durante o desenvolvimento, também é interessante usar o **Flutter Analyzer** com padrões do **Effective Dart** e corrigir *warnings* e *hints* indicados pela IDE.

- **Controle de Versão e DevOps:** Toda a colaboração no código será gerenciada via **Git**, usando uma plataforma como GitHub ou GitLab para hospedar o repositório do projeto. Recomenda-se adotar um fluxo de trabalho moderno, com *branches* para funcionalidades e *pull requests* para revisão (ver seção de boas práticas). Para integração contínua, pode-se configurar o **GitHub Actions** (ou similar) para automatizar builds e testes a cada push, garantindo que o app sempre compile e os testes passem [9]. Existem *actions* predefinidas para Flutter que rodam `flutter analyze` e `flutter test`. Ferramentas específicas para CI/CD mobile, como o **Codemagic**, também podem ser consideradas – o Codemagic oferece pipelines prontos para Flutter, incluindo geração de APK/IPA e até publicação automatizada em lojas. No contexto da disciplina, o enfoque é garantir que os alunos tenham contato com CI/CD: por exemplo, configurar ao

menos o pipeline de CI para *build* e testes unitários, e talvez simular um fluxo de entrega contínua gerando um APK de teste a cada iteração (que pode ser distribuído para o professor ou colegas avaliarem). A publicação final em lojas (Play Store/App Store) não é exigida, mas **gerar builds de distribuição (APK/Bundle)** e conhecer o processo de publicação faz parte da experiência – idealmente, na última semana, os alunos devem gerar um APK assinadoo ou usar TestFlight (se houver contas de desenvolvedor Apple) para demonstrar o app em ambiente próximo ao real.

Em suma, a pilha tecnológica sugerida inclui **Flutter/Dart** no front-end e **Supabase/Postgres** no back-end, complementados por bibliotecas de suporte conforme necessidade (gerência de estado, UI, APIs externas) e ferramental de desenvolvimento moderno (Git, CI, testes) – tudo alinhado às práticas atuais de desenvolvimento mobile e acessível para estudantes iniciantes em Flutter.

# 4. Boas Práticas a Adotar na Disciplina

Além das tecnologias em si, a disciplina deve enfatizar **boas práticas de engenharia de software** que serão esperadas em projetos mobile profissionais. Ao longo do desenvolvimento do projeto, os alunos deverão incorporar práticas que melhorem a organização, qualidade e manutenibilidade do código, bem como a eficiência do trabalho em equipe. Dentre as práticas recomendadas, destacam-se:

**Controle de Versão e Workflow Git:** Todo o código fonte do projeto deverá residir em um repositório Git (por exemplo, no GitHub da turma ou instituição). É importante definir um fluxo de trabalho claro com Git – por exemplo, pode-se adotar o modelo *Git Flow* (ramificação principal `main` ou `master`, branch de desenvolvimento `develop`, *feature branches* para novas funcionalidades, etc.) ou um modelo mais simples *trunk-based* com uso de *feature flags*. Cada nova funcionalidade ou correção deve ser desenvolvida em uma branch separada e então integrada via *pull request*, permitindo revisão de código pelos colegas ou professor antes de ser mesclada na branch principal [10]. Isso não só evita quebras na base de código, mas também promove troca de conhecimento e colaboração. É desejável usar mensagens de commit claras e semânticas, além de tags de versão (versionamento semântico) para marcar releases importantes do aplicativo (ex.: v1.0.0). Ainda que o projeto seja um só app (diferente do caso de múltiplos microserviços), se houver subprojetos (por exemplo, código de funções Edge separadas, ou protótipos), manter consistência de nomenclatura de branches e organização é crucial. De maneira geral, commits frequentes e bem documentados são encorajados, em vez de grandes dumps de código de última hora.

**Integração Contínua e Deploy Contínuo (CI/CD):** Configurar pipelines de integração contínua desde o início do projeto garante que a equipe receba feedback rápido sobre a saúde do código a cada mudança. Por exemplo, pode-se configurar o GitHub Actions para que a cada *push* ou *pull request* o projeto seja automaticamente construído (build) e os testes automatizados sejam executados [9]. Isso garante que erros de compilação ou testes quebrados sejam detectados imediatamente, facilitando correções enquanto o contexto da mudança ainda é recente. No contexto mobile, o pipeline pode rodar comandos como `flutter analyze` (linter), `flutter test` (testes unitários/widget) e `flutter build apk --debug` apenas para verificar se o build não falha. Já a **entrega contínua** em produção (CD) – como publicar o app automaticamente a cada commit na loja – não é viável no ambiente acadêmico (publicações móveis exigem revisões manuais, contas de desenvolvedor, etc.). Contudo, pode-se **simular** ou parcialmente implementar CD durante o curso: por exemplo, ao final de cada sprint importante (ver roteiro), gerar um *build* de teste do app e disponibilizá-lo para a turma/professor (via link do Firebase App Distribution, TestFlight interno, ou mesmo compartilhando o APK). Assim, os alunos se familiarizam com o conceito de entregar incrementos frequentes e aptos a deploy [11]. No mínimo, recomenda-se fazer deploy manual do aplicativo para um dispositivo de testes de forma

periódica (quinzenal, talvez) ao longo do desenvolvimento, em vez de deixar para integrar tudo somente no final. Essa prática ajuda a revelar problemas de integração (ex.: uma funcionalidade interfere em outra, ou algo que funcionava no emulador falha no dispositivo real) com antecedência.

**Arquitetura Interna e Organização do Código:** Desde o início, os alunos devem organizar o projeto Flutter com uma estrutura de pastas e código limpa, pensando em escalabilidade. É recomendável seguir um padrão arquitetural, como **MVVM (Model-View-ViewModel)** ou a **Arquitetura Limpa (Clean Architecture)** adaptada para Flutter, que separe responsabilidades em camadas. Por exemplo, pode-se dividir o código em: camada de UI (widgets/telas), camada de gerenciamento de estado (view models, BLoCs ou providers), camada de acesso a dados (serviços ou repositórios que comunicam com o Supabase) e modelos de dados (classes Dart representando objetos do domínio, p.ex. `Pedido`, `Usuario`, etc.). Essa separação torna o código mais **modular e de baixo acoplamento**, facilitando testes e manutenção. Cada classe/componente deve ter responsabilidade única bem definida (princípio de *single responsibility*). Evitar colocar lógica de negócio pesada diretamente em widgets; preferir que a UI apenas observe estados gerenciados por uma camada de lógica. Também é importante padronizar a estrutura de pastas – por exemplo, uma possível organização: `lib/screens/` (telas e widgets), `lib/models/` (modelos de dados), `lib/controllers/` ou `lib/viewmodels/` (caso MVVM, contendo lógicas), `lib/services/` (código de acesso Supabase ou outras APIs), etc. Consistência nessa estrutura entre equipes facilita todos navegarem no código. Durante o projeto, refatorações são bem-vindas: identificar trechos de código duplicado ou pouco elegantes e melhorar sua estrutura faz parte das boas práticas. Ferramentas como o analisador estático do Dart ajudarão a encontrar *code smells*. Além disso, utilizar convenções de nomenclatura adequadas (ex.: nomes de classes em PascalCase, nomes de variáveis e funções em camelCase, não usar abreviações confusas) e documentar partes complexas com comentários claros (mas evitar comentários desnecessários em código autoexplicativo) ajuda na legibilidade [12].

**Gerenciamento de Estado no Flutter:** Conforme mencionado na seção de tecnologias, adotar um padrão de gerenciamento de estado é fundamental. Como boa prática, todos os membros do grupo devem estar alinhados na solução escolhida (Riverpod, Bloc, etc.) para não misturar abordagens diferentes no mesmo projeto. Alguns princípios gerais: manter o *estado local* quando possível (por exemplo, estados efêmeros de um widget podem usar `setState` dentro dele mesmo), mas elevar (*lift state up*) estados que precisam ser compartilhados entre múltiplos widgets para um gerenciador central. Evitar o uso excessivo de variáveis globais ou singletons para armazenar estado, pois isso dificulta controle e testes. Se usar **Riverpod**, aproveitar providers para declarar estados globais ou dependentes de outros estados, facilitando a reatividade; por exemplo, um `StateNotifierProvider` para gerenciar a lista de itens no carrinho de compras e seu estado (lista vazia, carregando, erro, etc.). Com **Bloc**, dividir eventos e estados e reagir a eventos do usuário despachando-os ao Bloc apropriado. Independentemente da solução, focar em manter a UI reativa: ou seja, a interface deve reconstruir automaticamente quando os dados mudam (via `Consumer`/`Provider.of` no Riverpod/Provider, ou via `BlocBuilder` no Bloc, etc.), ao invés de depender de chamadas manuais para atualizar componentes. Isso previne uma classe de bugs comuns onde a UI exibe dados defasados. Outra boa prática é **desacoplar o código de interface do código de negócio**: por exemplo, um método que valida um login ou calcula um subtotal de pedido deve poder ser chamado/testado sem requerer elementos de UI. Essa lógica pode ficar em classes puras (no viewmodel ou bloc), enquanto a tela apenas chama métodos e exibe resultados. Para facilitar testes, evitar acessar diretamente `WidgetsBinding.instance` ou coisas contextuais dentro de lógicas – passar valores por parâmetro, usar Injeção de Dependência (ID) se necessário (ex.: no Riverpod é fácil injetar implementações simuladas de repositórios nos providers para teste). Em suma, **organização do estado e da lógica** é chave para que o aplicativo permaneça compreensível conforme cresce.

**Testes Automatizados:** A qualidade do aplicativo deve ser assegurada com testes em múltiplos níveis sempre que possível [13]. Deve-se enfatizar aos alunos escrever testes de unidade para as menores unidades de lógica – por exemplo, métodos utilitários (como cálculo de total, validação de CPF, fórmulas financeiras no app de finanças) e para componentes de negócio (por exemplo, verificar que ao adicionar um item ao carrinho, o estado do carrinho atualiza corretamente, ou que determinada consulta ao repositório retorna dados transformados como esperado). Além disso, testes de **widget** são valiosos para garantir que um determinado widget isolado se comporta e renderiza corretamente dado um estado ou entrada (por exemplo, passar uma lista de pedidos fictícios a um widget de lista e verificar se ele exibe todos os elementos; ou testar que um botão desabilita quando um form está inválido). Para funcionalidades críticas, recomenda-se praticar **Test-Driven Development (TDD)**: escrever primeiro testes que falham para a funcionalidade desejada e então codificar até que passem [14] – isso pode aumentar a confiabilidade e orientar um design de código melhor. Em seguida, testes de **integração (end-to-end)** devem ser realizados para cobrir fluxos completos do app: com o app rodando, simular interação do usuário e verificar se ele consegue, por exemplo, cadastrar-se, fazer login, inserir um dado e vê-lo refletido em outra tela, etc. Ferramentas como o `integration_test` do Flutter ou frameworks de terceiros podem auxiliar; alternativamente, pode-se realizar testes exploratórios manuais e capturar cenários de falha. É importante também testar condições de erro: como o app se comporta se o Supabase estiver offline (simular desligando a internet), ou se uma operação retorna erro (ex.: tentar salvar dados inválidos)? Tratar graficamente esses casos (mostrar mensagens ao usuário) e cobrir alguns via testes. Idealmente, integrar os testes à pipeline CI, para que sempre rodem automaticamente. Ao final do projeto, espera-se um mínimo de cobertura automática em funcionalidades centrais, complementado por um relatório de testes manuais realizados (testes de usabilidade, teste de aceitação seguindo as user stories definidas, etc.).

**Documentação Contínua:** Deve-se documentar o projeto ao longo de todo o desenvolvimento, não deixando para fazê-lo apenas na véspera da entrega [15]. A documentação serve tanto para o usuário final (se aplicável) quanto para desenvolvedores atuais e futuros entenderem as decisões tomadas. Alguns artefatos de documentação recomendados: - *Documento de Visão e Escopo:* já no início, formalizar o problema que o aplicativo se propõe a resolver, público-alvo, funcionalidades principais e requisitos (isso provavelmente será produzido na Semana 1-2 como entregue). - *Documentação de Arquitetura:* manter um documento (ou página wiki) descrevendo a arquitetura do sistema – neste caso, explicando a estrutura cliente/Supabase adotada, quais módulos existem no app, como o fluxo de dados acontece entre Flutter e Supabase, e quaisquer decisões arquiteturais importantes (por exemplo, "optamos por Riverpod em vez de Bloc por tais motivos..."). Podem ser usados *ADR (Architecture Decision Records)* para registrar decisões tecnológicas relevantes e suas justificativas [16]. Diagramas também ajudam: um diagrama simples de componentes mostrando o app, o Supabase, e integrações externas; modelo de dados Entidade-Relacionamento do banco; diagrama de navegação de telas; etc. - *Documentação de API/Dados:* embora não haja uma API custom desenvolvida (usamos a do Supabase), é útil documentar os principais **endpoints/queries** que o app faz e a estrutura dos dados envolvidos. Por exemplo, detalhar as tabelas principais (campos, tipos, restrições) e como elas se relacionam. Se fossem usadas Edge Functions, documentar suas rotas e payloads esperados. Isso serve como referência para a equipe e para eventuais testers. Poderia-se usar uma ferramenta de documentação (como Swagger/OpenAPI) adaptada para descrever endpoints do Supabase, mas uma simples tabela ou markdown listando "Tabela X: campos e suas descrições" já ajuda bastante. - *Manual do Usuário:* conforme o aplicativo fique pronto, produzir um guia básico de uso, com capturas de tela, mostrando como realizar as principais tarefas. Isso é útil para apresentar ao professor e colegas na entrega final, e demonstra preocupação com a experiência do usuário. - *README e Comentários no Código:* Manter um README atualizado no repositório, com instruções de como rodar o projeto (prérequisitos de ambiente, comandos de build, variáveis necessárias como URL do Supabase), bem como resumo do projeto. No código, usar comentários apenas para explicar partes complexas ou algoritmos não triviais – código bem estruturado deve ser compreensível sem necessitar comentários extensivos. Adotar um guia de

estilo consistente (por exemplo, **Effective Dart** para estilo de código Dart) e utilizar ferramentas de lint/formatter (o Flutter já vem com `flutter format` e recomendações de lint pelo pacote `flutter_lints`) para padronizar a formatação do código automaticamente [12]. Isso evita divergências de estilo entre desenvolvedores e torna o código mais limpo.

**Gerenciamento de Configuração e Segurança:** Atenção especial deve ser dada a configurações sensíveis do app, como chaves e URLs. **Nunca incluir credenciais sensíveis em texto plano no código ou no controle de versão** [17]. No caso do Supabase, a chave anônima pública pode ficar no app (pois ela sozinha não compromete segurança graças ao RLS), mas a URL do backend e quaisquer chaves privadas não devem vazar. Uma boa prática é utilizar um arquivo de configuração separado ou mesmo variáveis de ambiente (há pacotes como `flutter_dotenv` que permitem carregar configurações de um arquivo .env não enviado ao repositório). Manter diferentes configurações por ambiente – exemplo: apontar para um projeto Supabase de desenvolvimento durante o desenvolvimento, e para outro projeto Supabase (produção) se forem fazer uma demonstração final com dados "limpos". Gerar um arquivo `env.sample` no repo com as chaves/variáveis necessárias (mas não os valores reais) para que todos saibam o que configurar. Além disso, considerar medidas de segurança no app: **validar inputs** do usuário (mesmo que o backend também valide, é bom prevenir dados incorretos na fonte), tratar exceções de forma que o app não quebre (e idealmente reportar erros de runtime – talvez integrando um serviço como Sentry ou Firebase Crashlytics para log de erros, se houver tempo). Garantir também que, ao fazer requisições ao Supabase, erros de autenticação sejam tratados (ex.: token expirado – o app deve redirecionar para login se necessário), e que dados sensíveis do usuário (como senhas) nunca sejam armazenados de forma insegura. Usar sempre comunicação HTTPS (padrão do Supabase) e certificados válidos. Embora algumas dessas preocupações sejam avançadas, introduzi-las na disciplina prepara os alunos para desenvolverem aplicativos robustos e seguros.

Em resumo, as boas práticas englobam organização disciplinada do trabalho (Git, CI/CD), organização limpa do código e arquitetura, uso criterioso de ferramentas de estado, garantia de qualidade via testes e documentação contínua, e atenção à segurança e configurações. Ao seguir essa estrutura, os alunos estarão **experimentando um desenvolvimento mobile profissional** em miniatura, o que enriquece o aprendizado muito além da codificação básica.

## 5. Roteiro de Artefatos e Entregas ao Longo da Disciplina

Para garantir o progresso organizado do projeto durante o semestre, sugere-se um cronograma de entregas (sprints) com marcos e artefatos específicos. Abaixo um roteiro possível (que pode ser ajustado conforme a duração exata do curso e carga horária):

**Semana 1-2: Definição do Escopo e Requisitos – Entrega: Documento de Visão e Escopo.** Nos primeiros encontros, os alunos (em equipes ou individualmente) devem **escolher o tema** da aplicação móvel (dentre as opções sugeridas ou outra proposta aprovada pelo professor) e levantar os **requisitos de alto nível**. Isso inclui uma descrição do problema a ser resolvido e do público-alvo, principais funcionalidades desejadas, atores (tipos de usuários do aplicativo) e algumas **user stories** exemplares descrevendo interações típicas. Por exemplo: "Como um cliente de delivery, quero conseguir visualizar restaurantes próximos e fazer um pedido para receber comida em casa". Também devem ser listados requisitos não-funcionais relevantes, como portabilidade (Android/iOS), desempenho (tempo de resposta desejado), segurança (proteção de dados financeiros, etc.). Como parte desse artefato inicial, espera-se uma **lista priorizada de funcionalidades** que definem o MVP (produto mínimo viável) do aplicativo. Além disso, já nesta fase define-se a tecnologia a ser usada em cada parte (Flutter no front, Supabase no back, bibliotecas de estado, etc.), alinhando com as recomendações dadas pelo professor

e o plano aqui. *(Dica: utilizar o método CERTO – ver seção 6 – para consultar o ChatGPT e obter ideias iniciais de funcionalidades ou validação de requisitos, fornecendo o contexto do projeto e exigências do curso* [18] *.)*

**Semana 3-4: Modelagem de Dados e Design da Arquitetura – Entrega: Modelo Conceitual e Desenho Arquitetural.** Com o escopo definido, os alunos partem para planejar a solução em termos de **estrutura de dados** e **arquitetura do app**. Devem elaborar diagramas e documentos que detalhem a solução proposta. Espera-se aqui: - Um **Modelo Conceitual** (por exemplo, um Diagrama Entidade-Relacionamento ou Diagrama de Classes simples) representando as principais entidades do domínio e seus relacionamentos. Para o app de delivery, por exemplo: entidades Usuário, Pedido, Produto, Restaurante, etc., com atributos chave e ligações (um Restaurante tem muitos Produtos, um Pedido pertence a um Usuário e inclui vários Produtos, etc.). Esse modelo servirá de base para a criação das tabelas no Supabase. - Um **Diagrama de Arquitetura** da aplicação móvel, ilustrando os componentes principais e suas interações. Poderia ser um diagrama de componentes mostrando: app Flutter (dividido em camadas UI, State, Serviços), Supabase (Auth, Database, Storage, Edge Functions) e quaisquer serviços externos integrados. O objetivo é visualizar a divisão cliente/servidor e onde cada responsabilidade está. Opcionalmente, diagramas adicionais podem ser apresentados, como um **Diagrama de Fluxo de Navegação** entre telas do app, ou um **Diagrama de Sequência** ilustrando um fluxo end-to-end (por ex., "usuário faz um pedido" – mostrando as interações entre usuário (app) → banco de dados (Supabase) → notificações, etc.). - Além dos diagramas, um breve documento textual descrevendo as decisões de arquitetura – por que certas tecnologias/padrões foram escolhidos, como serão tratados requisitos não-funcionais (ex.: "para garantir desempenho, usaremos realtime ao invés de polling"; "todos os dados sensíveis terão RLS ativado no banco"; "optamos por Riverpod para gerenciamento de estado pelas seguintes vantagens…").

Nesta entrega também já se define a **estrutura de repositório** e configura-se o projeto base. Ou seja, os alunos deverão criar o projeto Flutter inicial (usando `flutter create`), aplicar a estrutura de pastas decidida, e integrar com o controle de versão (repositório Git). Se possível, configurar uma pipeline de CI inicial (nem que seja só para rodar um build/teste dummy) para já praticar DevOps. No Supabase, pode-se já criar o projeto no dashboard online e talvez provisionar as tabelas iniciais conforme o modelo concebido (isso pode ser feito via interface gráfica ou script SQL). Em suma, ao fim da semana 4 deve existir um *esqueleto* da aplicação: telas ainda vazias ou de exemplo, mas com projeto configurado; e um backend preparado (mesmo que ainda sem dados reais). Esta etapa garante que a arquitetura proposta é viável e que todos têm um ambiente funcional para começar a codificação – por exemplo, validar que o app consegue se conectar ao Supabase (fazer um "Hello World" autenticado, ou retornar algo mínimo do banco).

**Semana 5-6: Protótipo de Interface e Contratos de API/Dados – Entrega: Protótipo Navegável da UI e Especificações de Dados.** Enquanto parte da equipe (ou do tempo) pode iniciar alguma codificação básica, é altamente recomendável dedicar estas semanas para acertar o **design da interface** e os **contratos de comunicação** do app antes de mergulhar no desenvolvimento completo. Os alunos devem produzir **protótipos das telas principais**, de preferência usando uma ferramenta de design como **Figma** para criar um protótipo navegável de alta fidelidade das principais telas e fluxos do aplicativo (ex.: tela de login, tela principal/listagem, tela de detalhes, tela de formulário de novo item, etc., conforme o app). Esse protótipo permite avaliar a UX/UI, coletar feedback cedo e alinhar expectativas do que será implementado. Em paralelo, deve-se **definir os contratos de API/dados** entre front-end e back-end: por exemplo, especificar quais *endpoints* ou operações o app irá chamar no Supabase, quais parâmetros e retornos. Como o Supabase expõe diretamente operações no banco, isso pode ser descrito em termos de tabela/operação: "Obter lista de restaurantes: chamada GET na tabela `restaurants` filtrando por região do usuário; Retornar campos X, Y, Z…". Pode-se montar uma pequena documentação estilo REST para essas operações, ou mesmo usar a interface do **Supabase (Swagger)** para extrair a definição das rotas. Caso alguma lógica seja delegada a Edge Functions,

documentar seu endpoint HTTP, corpo JSON esperado e resposta. Essa especificação serve como contrato para que front e back avancem em paralelo [19] – por exemplo, o time de front pode desenvolver a tela de listagem assumindo que haverá um endpoint "listarProdutos" que retorna certos campos, enquanto o time de back configura o banco para garantir que esses campos estarão disponíveis e seguros. Além disso, pode ser apresentado aqui um **Diagrama de Sequência** focado em integrações, ilustrando como uma requisição do app interage com o backend: por exemplo, "Usuário posta um comentário" – sequência: App Flutter envia request → Supabase (Tabela comments) insere registro → Supabase Realtime notifica usuários online → App atualiza feed em tempo real. Isso ajuda a validar se todos entendem o fluxo completo dos dados. Ao final da semana 6, os artefatos esperados são: arquivos de design (Figma) ou screenshots do protótipo mostrando as telas principais navegáveis, e um documento/tabela descrevendo as APIs/operações de dados previstas para o MVP. A combinação desses artefatos funciona como um guia de implementação para as próximas etapas.

**Semana 7-9: Implementação – Iteração 1 (MVP Parcial) – Entrega: Versão MVP Integrada (parcial).** Inicia-se aqui a codificação intensa para construir uma primeira versão funcional mínima do aplicativo, integrando front-end e back-end. O objetivo ao final da semana 9 é ter um **MVP (Minimum Viable Product)** cobrindo as **funcionalidades essenciais** do aplicativo, ainda que com escopo reduzido e interface simples. Por exemplo, no caso do app de delivery, o MVP poderia permitir: cadastro/login de usuário, visualização de uma lista de produtos ou restaurantes (cadastrados manualmente ou previamente no banco), adicionar um produto ao carrinho e efetuar um pedido básico. Não precisa estar tudo completo ou bonito – o importante é provar o funcionamento end-to-end: dados sendo salvos no Supabase e recuperados no app, autenticação funcionando, etc. Isso provavelmente exigirá que pelo menos 2-3 **funcionalidades núcleo** estejam implementadas (no exemplo: autenticação de usuários, listagem de itens, criação de pedido) e que o front-end seja capaz de chamar o back-end apropriadamente. O trabalho deve incluir também a configuração inicial do banco de dados no Supabase (se ainda não finalizada): criação das tabelas definitivas, funções ou *triggers* se necessárias, e políticas de segurança RLS para proteger os dados. Desde essa primeira versão, é recomendável já ativar algum cenário de **Realtime** se aplicável – por exemplo, no app de rede social, ao postar um novo conteúdo a lista de posts dos usuários conectados atualiza em tempo real; ou no app de delivery, se possível, notificar um "novo pedido" numa interface de entregador (mesmo que simulada). Também, **configurar autenticação básica** no app (ex.: proteger rotas/telas para que somente usuários logados acessem, armazenar o token de sessão com segurança). Os alunos devem se esforçar para escrever **testes automatizados** pelo menos para a lógica central implementada nessa iteração (ex.: teste de criação de pedido atualiza estoque, teste de cálculo de orçamento restante no app de finanças). A entrega consistirá do código-fonte atualizado no repositório (espera-se que o repositório já contenha o histórico de commits mostrando o desenvolvimento incremental), instruções claras para rodar o app (por exemplo, qualquer configuração de ambiente necessária) e uma pequena demonstração em aula do MVP funcionando. Os alunos devem relatar também quais testes já implementaram e quaisquer problemas encontrados durante a iteração. Após essa entrega, espera-se **feedback do professor** para ajustes – por exemplo, correções de bugs identificados, melhorias de UX, ou revisão de alguma modelagem de dados que possa precisar de refinamento [20] .

**Semana 10-12: Iteração 2 (Funcionalidades Avançadas e Refinamento) – Entrega: App Completo em Ambiente de Teste.** Na segunda fase de desenvolvimento, os alunos expandem as funcionalidades para cobrir **todo o escopo planejado** inicialmente e refinam aspectos importantes do aplicativo. Isso inclui: - Implementar quaisquer **módulos restantes** que não foram contemplados no MVP. Por exemplo, no app de delivery: módulo de pagamentos (mesmo que simulado, integrando com um sandbox ou registrando pagamento fictício), módulo de avaliação/feedback, perfil do usuário (editar dados, histórico de pedidos) etc. No app de finanças: funcionalidade de orçamento mensal, gráficos detalhados, exportar dados; na rede social: sistema de comentários, chat privado, etc. Cada equipe seguirá aqui as prioridades definidas no backlog, visando entregar um produto funcional completo. -

**Tratar regras de negócio mais complexas** em cada funcionalidade. Por exemplo, aplicar validações robustas (não permitir cadastro duplicado, prevenir que um usuário gaste além do orçamento definido, etc.), e garantir fluxos alternativos (ex.: recuperação de senha na autenticação, cancelamento de pedido em andamento, edição e deleção de itens). - **Melhorar a robustez e desempenho** do aplicativo. Esta é a fase de *hardening*: adicionar tratamentos de erro abrangentes (mostrar mensagens amigáveis se algo falhar, fallback para ações offline se fizer sentido, etc.), otimizar partes lentas (por ex., implementar caching local de dados raramente mutáveis, como uma lista de categorias de produto, para evitar chamadas redundantes). Se durante os testes do MVP observaram problemas de performance, aqui é hora de ajustar – talvez usar consultas mais eficientes no Supabase (filtrar no banco em vez de trazer tudo e filtrar no app), usar índices no banco para campos muito buscados, paginar resultados em listas longas para não sobrecarregar o app, entre outros. Caso o app tenha muitas imagens, implementar lazy loading e usar thumbnails para não travar a UI. - **Adicionar caching/local storage** conforme necessário. Exemplo: no app de finanças, pode-se armazenar em um banco de dados local (using `Hive` ou `Floor`/`Sqflite`) os últimos lançamentos para consulta offline e sincronizar quando online – essa funcionalidade offline poderia ser um diferencial, embora não obrigatória. No app de delivery, talvez cachear os dados do restaurante para acesso rápido. Utilizar o package `shared_preferences` para pequenas preferências do usuário (tema claro/escuro, últimas opções usadas) também enriquece o aplicativo. - **Aprimorar a segurança e autenticação:** Verificar se todas as operações críticas estão protegidas. Por exemplo, assegurar que regras RLS no Supabase estão cobrindo todos os casos (testar tentando acessar dados de outro usuário para confirmar que o acesso é negado). No app, garantir que tokens de sessão expirados sejam tratados (talvez implementar refresh de token ou re-login suave). Remover quaisquer credenciais ou dados sensíveis expostos indevidamente no código (chaves de API, etc.), preparando o app para uma possível publicação pública. - **UX Polishing:** Lapidar a experiência do usuário – melhorar layout, incluir animações/transições onde couber, tornar a interface intuitiva. Corrigir quaisquer inconsistências de design apontadas no feedback do professor ou dos colegas. Garantir acessibilidade básica (tamanhos de fonte adequados, contraste de cores suficiente, labels em componentes para leitores de tela se for abordado). - **Testes adicionais:** Ampliar a suíte de testes automatizados para cobrir os novos cenários e funcionalidades adicionadas. Se antes apenas casos "felizes" foram testados, incluir também cenários de erro/limite. Medir novamente a cobertura de testes. Se o tempo permitir, realizar um pequeno **teste de carga** no back-end – por exemplo, usar scripts para simular múltiplos usuários fazendo operações simultaneamente e monitorar se o Supabase aguenta (dentro dos limites de quotas da conta). - **Beta Test (opcional):** Se possível, distribuir a versão quase final do app para alguns usuários testers (colegas de turma ou voluntários) para obter feedback de usabilidade real e detectar bugs que não apareceram nos testes internos. Esse beta poderia ser distribuído via um APK ou TestFlight.

Ao final da semana 12, o objetivo é ter **todas as features implementadas** conforme o escopo inicial definido. A entrega seria uma nova versão do aplicativo, preferencialmente disponibilizada em um ambiente de *staging* (por exemplo, um APK de release para Android enviado ao professor/testers, e se viável um TestFlight para iOS). Junto com o app, entregar também um conjunto ampliado de **testes automatizados** e um breve relatório de progresso, destacando o que foi adicionado ou mudado em relação ao MVP, e quaisquer alterações de arquitetura necessárias (por ex.: "descobrimos necessidade de adicionar uma tabela X para suportar funcionalidade Y"). Essa documentação ajuda o professor a acompanhar a evolução e entender as decisões tomadas.

**Semana 13: Testes Finais e Qualidade – Entrega: Relatório de Testes e Ajustes Finais.** Com o aplicativo funcional completo, esta semana é reservada para **testes integrados completos e garantia de qualidade** antes do *deploy* final. Os alunos devem executar um conjunto abrangente de **testes end-to-end**, cobrindo todos os requisitos e histórias de usuário levantadas. Idealmente, envolver pessoas que não participaram diretamente do desenvolvimento para testar (para ter um olhar fresco, se possível). Cada funcionalidade deve ser verificada em cenário real: por exemplo, no app de delivery,

realizar um fluxo completo desde cadastro até finalizar um pedido, verificar se notificações chegam, etc. É útil montar uma checklist ou planilha de casos de teste (incluindo cenários de erro) e marcar quais passaram ou falharam, anotando bugs a corrigir. Também é importante **monitorar desempenho básico**: medir o tempo de resposta das principais operações (login, carregar lista, salvar item) e assegurar que estão aceitáveis (ex.: app de social feed carregando em <2s em boa conexão). Se possível, testar o comportamento do app com diferentes condições: com internet lenta ou oscilante, em dispositivos com diferentes tamanhos de tela, e simulando falhas do back-end (por ex.: desligar intencionalmente a rede e ver se o app lida com isso graficamente). Qualquer bug crítico encontrado deve ser corrigido nesta fase. Além disso, verificar logs de erro (se incluíram Crashlytics ou Sentry, conferir se algo foi registrado durante os testes). Os critérios de aceitação definidos lá no começo (Semana 1-2) devem ser revisitados para garantir que todos foram atendidos no produto final. O artefato de saída dessa semana é um **Relatório de Testes**, contendo: resumo da cobertura de testes automatizados (% de código coberto, número de casos de teste); resultados dos testes manuais (lista de casos testados e status, lista de bugs encontrados e resolvidos); e eventuais *feedbacks* de usabilidade coletados e ajustes realizados para melhorar a experiência do usuário. Esse relatório demonstra o nível de qualidade atingido e serve como evidência de validação. Se algum requisito acabou não sendo atendido, deve ser justificado aqui. Ao final da semana 13, o aplicativo deve estar em estado de **release candidate** – pronto para ser apresentado e entregue, sem bugs conhecidos de grande impacto.

**Semana 14-15: Documentação Final e Publicação – Entrega: Pacote Final do Projeto.** Na etapa derradeira, os alunos consolidam toda a documentação e preparam a apresentação/entrega final do sistema. Deve-se compilar: - **Documentação do Usuário:** um manual ou guia breve explicando como usar o aplicativo, com capturas de tela da versão final e instruções passo a passo para funcionalidades principais. Se o app for publicado (mesmo que em beta), incluir link ou QR code para instalação. - **Documentação do Desenvolvedor:** detalhes de como rodar o projeto (configuração de ambiente, instalação de dependências, como configurar variáveis de ambiente do Supabase), estrutura de pastas do código, explicação de componentes importantes e quaisquer peculiaridades para manutenção futura. Incluir aqui também os diagramas finais de arquitetura e modelo de dados atualizados caso tenham mudado desde a fase de design inicial. - **Código-Fonte Final:** garantir que todo o código está devidamente versionado no repositório, com uma *tag* de versão final (ex.: v1.0.0). Incluir no repositório todos os artefatos relevantes (documentação pode estar no README ou em arquivos PDF no repo). - **Builds para Demo:** preparar a **distribuição do app** para a apresentação final. Para Android, gerar um APK ou AppBundle assinado de release. Para iOS (se aplicável e se houver conta de desenvolvedor), distribuir via TestFlight ou providenciar o arquivo .app/.ipa instalável em um dispositivo. O ideal é que na apresentação cada equipe consiga rodar sua aplicação em dispositivos reais para demonstrar (ou emuladores, caso necessário), então verificar antecipadamente a compatibilidade. - **Dados de Demo:** opcionalmente, preparar um conjunto de dados fictícios porém realistas no banco (ex.: no app de delivery, cadastrar 5 restaurantes de exemplo com produtos e 2-3 usuários de teste; no app de finanças, popular com algumas transações de exemplo) para tornar a demonstração mais interessante sem precisar inserir tudo na hora. Esses dados podem ser inseridos via script SQL no Supabase ou manualmente pela interface.

Nesta fase final, todos os diagramas e documentos devem estar refinados e consistentes com a implementação atual do sistema (por exemplo, se alguma entidade ou fluxo mudou durante o desenvolvimento, atualizar o diagrama correspondente elaborado lá atrás). A **apresentação final** ocorrerá ao término da semana 15 (ou 16, dependendo do calendário): cada equipe mostra seu aplicativo funcionando, destaca os principais desafios e soluções, e responde a perguntas. A entrega formal consiste no pacote completo contendo o código, documentação e artefatos produzidos. Assim, encerra-se a disciplina com cada grupo tendo desenvolvido e documentado uma aplicação móvel funcional utilizando Flutter e Supabase, passando por todas as etapas desde concepção até publicação simulada.

# 6. Aplicação do Método CERTO com IA no Projeto

Para potencializar o planejamento e a execução do projeto, propõe-se integrar o uso de **Inteligência Artificial** (ex: ChatGPT) como ferramenta de apoio, utilizando o método **CERTO** para elaborar *prompts* eficazes. Desenvolvido por Thais Martan, o método CERTO sugere estruturar cada consulta à IA com **cinco elementos – Contexto, Exigências, Referências, Tarefas e Observações** – de forma a fornecer o máximo de clareza e informação ao modelo [21]. Seguindo essa estrutura, garante-se que a ferramenta (IA) receba todos os dados necessários para gerar respostas mais assertivas e úteis, atuando como uma espécie de "assistente" durante o desenvolvimento.

Como aplicar na disciplina? Ao longo do projeto, os alunos (ou o professor) podem empregar o método CERTO em diversas situações, tais como: **planejamento de funcionalidades**, **resolução de problemas de código**, ou **busca de sugestões de arquitetura**. Por exemplo:

- **Planejamento do Escopo (Início do Projeto):** Durante a definição de requisitos e brainstorming inicial, os estudantes podem consultar o ChatGPT para obter ideias adicionais ou validar o escopo pretendido. Usando CERTO, eles devem fornecer **Contexto** (descrição do projeto escolhido, público-alvo, objetivos principais), listar as **Exigências** (por ex.: "a aplicação deve ser construída em Flutter com Supabase, deve ser concluída em 4 meses por uma equipe de 5 alunos, precisa rodar offline e online"), incluir **Referências** (ex.: "temos como referência apps X e Y semelhantes" ou listar requisitos já definidos, e até links de matéria dada em aula), definir claramente a **Tarefa** que a IA deve realizar ("Gerar 5 ideias de funcionalidades inovadoras para nosso app de finanças pessoais que aproveitem recursos de realtime e engajamento do usuário") e adicionar **Observações** ou preferências (ex.: "responder em português", "considerar apenas tecnologias open-source"). Essa abordagem estruturada orienta a IA a dar sugestões focadas e viáveis, atuando como um *brainstorming* assistido [22] [23]. Os alunos podem, a partir das sugestões, avaliar quais se encaixam melhor no projeto e incorporá-las ao backlog.

- **Design da Arquitetura:** Na fase de arquitetura, caso os alunos queiram validar o desenho proposto ou buscar melhorias, podem preparar um prompt descrevendo o design atual do sistema. Por exemplo, fornecer o **Contexto**: "Desenvolvemos um app de delivery usando Flutter no front e Supabase no back"; as **Exigências** ou restrições: "precisamos suportar 1000 usuários, manter baixo custo, garantir segurança dos dados (RLS)"; as **Referências**: "estamos seguindo uma arquitetura MVVM no app e utilizamos Riverpod, e estudamos o padrão de microsserviços na disciplina Sistemas Corporativos"; então definir a **Tarefa**: "Avaliar nossa arquitetura e sugerir possíveis melhorias ou pontos fracos, especialmente focando em desempenho e escalabilidade do app"; e Observações finais: "nossa arquitetura atual inclui realtime para atualizações, preferimos não adicionar serviços externos pagos". Com essa estrutura, a IA, guiada por essas informações, pode apontar melhorias arquiteturais (por ex.: recomendar uso de caching local para reduzir chamadas, ou separar melhor responsabilidades em camadas) ou confirmar que o caminho adotado está coerente, sempre embasando a resposta nos dados fornecidos [24].

- **Implementação e Codificação:** Durante o desenvolvimento, é comum surgirem dúvidas de sintaxe, erros de código ou necessidade de exemplos. Os alunos podem usar o ChatGPT como um "par" de programação virtual para superar esses obstáculos. Ao escrever um prompt, devem sempre estruturar em CERTO. Por exemplo: **Contexto**: "Estou desenvolvendo a tela de login do app em Flutter"; **Exigências**: "Preciso usar o pacote flutter_bloc para gerenciar estado, e implementar validação de email e senha conforme regras X (ex.: senha 8+ caracteres). Quero que ao logar, salve o token JWT do Supabase com segurança"; **Referências**: "Segue um trecho do código atual que não está funcionando como esperado:" (colando o código Dart ou a mensagem

de erro recebida); **Tarefa**: "Identifique o erro no código e sugira correções ou melhorias seguindo as boas práticas do Flutter"; **Observações**: "O erro ocorre quando chamo a função de login, já tentei uma solução usando try/catch mas não obtive sucesso". Ao fornecer esses detalhes, a IA consegue analisar com mais precisão o cenário específico, muitas vezes detectando erros lógicos ou de sintaxe que passaram despercebidos, e sugerindo trechos de código corretos ou mais eficientes [25] [26]. Essa dinâmica se assemelha a ter um colega experiente revisando seu código e dando dicas.

- **Geração de Testes e Documentação:** Outra aplicação útil da IA é ajudar na produção de testes automatizados ou na redação de partes da documentação. Por exemplo, os alunos podem pedir: "Sugira casos de teste unitário para a classe X (descrevendo o que a classe faz)" – fornecendo **Contexto** da funcionalidade, **Exigências** (cobrir casos de sucesso e falha, considerar limites, etc.), eventuais **Referências** (mostrando a implementação atual da classe) e a **Tarefa** ("Gerar 5 cenários de teste"). A IA pode retornar ideias de testes relevantes que os alunos talvez não tenham lembrado, servindo como ponto de partida para escreverem seu código de teste [27]. De modo similar, para a documentação: "Ajude a escrever a documentação do endpoint Y" passando contexto do endpoint, referência do formato de resposta esperada, etc., e a tarefa "Gerar uma descrição clara e exemplos de uso". A IA produz um rascunho que os alunos podem refinar, agilizando tarefas repetitivas.

- **Resolução de Problemas e Depuração:** Quando os alunos enfrentarem um bug persistente ou um bloqueio técnico, podem articular o problema para a IA seguindo CERTO. Por exemplo: **Contexto**: "Ao tentar salvar um novo pedido, a aplicação está lançando uma exceção"; **Exigências**: "Resolver sem quebrar a funcionalidade de cálculo de estoque, e de preferência sem remover as políticas RLS do Supabase"; **Referências**: "Log de erro:" (colam o stack trace ou mensagem de erro) e talvez o trecho de código relevante; **Tarefa**: "Encontrar a causa raiz do bug e sugerir uma solução"; **Observações**: "Esse erro começou após implementarmos a checagem de duplicatas, já tentamos reverter essa parte mas o erro persiste no ambiente Android". Muitas vezes, somente o ato de estruturar o problema desse modo já ajuda os alunos a organizarem o pensamento e visualizar hipóteses de solução. A IA, por sua vez, pode identificar o *ponto cego* rapidamente ou propor técnicas de depuração adicionais [28] que os alunos ainda não tentaram.

O importante é frisar que o uso do método CERTO deve ser incentivado como uma **ferramenta de apoio**, não para fazer o trabalho dos alunos, mas para **ampliar a capacidade de pesquisa e resolução** deles. O professor pode inclusive requerer que, em relatórios, os alunos citem se usaram a IA para determinada tarefa e como foi a experiência, fomentando discussão crítica sobre os resultados. Isso torna a disciplina inovadora ao integrar IA no fluxo de trabalho de desenvolvimento de software, algo cada vez mais comum na indústria [29].

Em suma, o método **CERTO** (*Contexto, Exigências, Referências, Tarefas, Observações*) fornece um guia para criar prompts claros e eficazes, garantindo que a IA entenda exatamente o que se precisa. Conforme descrito por Martan, seguir essa estrutura mnemônica aumenta a assertividade das respostas do ChatGPT [30]. Aplicado ao projeto, ele pode auxiliar no planejamento e na execução, desde a concepção do sistema até a depuração final, sempre de forma ética e construtiva. Essa integração do método CERTO no curso também prepara os alunos para usar ferramentas de IA de maneira profissional e responsável, uma competência cada vez mais valorizada no mercado de desenvolvimento de sistemas [31].

**Referências Utilizadas:** As recomendações e melhores práticas aqui descritas foram baseadas em conteúdo de referência, incluindo o plano da disciplina de Sistemas Corporativos [32] [33], documentação oficial do Supabase [3] [34], tutoriais de Flutter com Supabase [2] [4], e diretrizes

compartilhadas pela comunidade de Flutter/Dart [8] [10], entre outros. A metodologia CERTO foi apresentada conforme Martan [21], com exemplos adaptados ao contexto mobile. Essas fontes enriquecem o plano com sugestões atualizadas e alinhadas às práticas de mercado.

---


[1] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] 01.pdf
file://file-E2uFKruKVkq7GSwRyQmhv5

[2] [4] Flutter Tutorial: building a Flutter chat app
https://supabase.com/blog/flutter-tutorial-building-a-chat-app

[3] [5] [6] [34] Supabase | The Postgres Development Platform.
https://supabase.com/

[7] Top 5 Flutter UI Libraries for 2024 (Updated) - GeekyAnts
https://geekyants.com/en-us/blog/top-5-flutter-ui-libraries-for-2023

[8] Top Flutter Widget Library and UI Framework packages
https://fluttergems.dev/widget-library-ui-framework/