**Updated Angular Frontend Prompts **

1. Prompt: Separation of Concerns (SRP)

~~~
**Goal:**
Analyze the `dashboard.component.ts` file to determine if it adheres to t

**Context:**
You are an Angular architecture review agent. A "lean" component should c

**Source Code to Analyze:**
```typescript

// src/app/features/dashboard/dashboard.component.ts
~~~

Expectations:

1. **Verdict:** `Pass` or `Fail` .
2. **Score (1-10):** Rate the component's adherence to SRP.
3. **Evidence:** Provide a code snippet that shows business logic being handled directly within the component (Fail).
4. **Suggestion:** If `Fail` , recommend refactoring the identified business logic into a dedicated service (e.g., `DashboardService`).

~~~
#### **2. Prompt: Naming Conventions**
```markdown
**Goal:**
Audit the `dashboard` feature's file and symbol names for adherence to the
official Angular style guide.

**Context:**
You are a code consistency agent. The Angular style guide prescribes
specific naming conventions:
-   File names must be lowercase and use kebab-case (e.g.,
`dashboard.component.ts`).
-   Class and symbol names should follow the `[Feature].[Type]` pattern
and use PascalCase with the correct suffix (e.g., `DashboardComponent`,
`AuthGuard`).
~~~

```
**Source Code to Analyze:**
Please review the following file names and the class names within them:
-   `src/app/features/dashboard/dashboard.component.ts`
-   `src/app/features/dashboard/dashboard.component.html`
-   `src/app/features/dashboard/dashboard.component.scss`
-   `src/app/features/dashboard/dashboard.routes.ts`

**Expectations:**
1.  **Verdict:** `Pass` or `Fail`.
2.  **Score (1-10):** Rate the overall consistency of naming conventions.
3.  **Evidence:** List any file or class names that violate the
conventions.
4.  **Suggestion:** Provide a specific list of files or classes that need
to be renamed to conform to the style guide.
```

3. Prompt: Template Cleanliness (No Logic)

~~~
**Goal:**
Review the `dashboard.component.html` template to ensure it is purely dec

**Context:**
You are a template review agent. Angular templates should be simple and r

**Source Code to Analyze:**
```html

// src/app/features/dashboard/dashboard.component.html
~~~

Expectations:

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate the cleanliness of the template.
3. **Evidence:** Quote a line from the template that contains complex logic.
4. **Suggestion:** If `Fail`, recommend moving the logic into a new method or property on the component's `.ts` file.

~~~
#### **4. Prompt: RxJS and Memory Management**
```markdown
**Goal:**
Audit the `dashboard.component.ts` for potential memory leaks by checking
~~~

for unmanaged RxJS subscriptions.

Context:
You are a memory management agent. Manually subscribing to an Observable (`.subscribe(...)`) without a proper teardown mechanism (like `takeUntil`) is a common source of memory leaks. The best practice is to use the `async` pipe in the template.

Source Code to Analyze:
~~~typescript

// src/app/features/dashboard/dashboard.component.ts

**Expectations:**

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate the safety of the RxJS subscription handling.
3. **Evidence:** Provide a code snippet showing a `.subscribe()` call that is not properly unsubscribed in `ngOnDestroy`.
4. **Suggestion:** If a potential leak is found, recommend implementing the `takeUntil(this.destroy$)` pattern.

#### **5. Prompt: Immutability & Change Detection**
```markdown
**Goal:**
Verify that a presentational component from your `shared` folder uses `ChangeDetectionStrategy.OnPush` and treats data inputs as immutable.

**Context:**
You are a performance optimization agent. For better performance, presentational components (like a generic card or button) should use the `OnPush` change detection strategy. This requires that all `@Input()` data is treated as immutable.

**Source Code to Analyze:**
```typescript

// src/app/shared/components/card/card.component.ts

**Expectations:**

1. **Verdict:** `Pass` or `Fail`.
~~~

2. **Score (1-10):** Rate the implementation of `OnPush` and immutability.
3. **Evidence:** Quote the `@Component` decorator.
4. **Suggestion:** If not present, recommend adding `changeDetection: ChangeDetectionStrategy.OnPush` to the component decorator.

~~~~
#### **6. Prompt: Performance Optimizations (`trackBy`)**
```markdown
**Goal:**
Check if the `dashboard.component.html` template uses the `trackBy`
function for any `*ngFor` loops that iterate over lists of objects.

**Context:**
You are a DOM performance agent. When an array is modified, Angular re-
renders the entire DOM list by default. Providing a `trackBy` function
tells Angular how to track each item, which is a critical performance
optimization for lists.

**Source Code to Analyze:**
```html

// src/app/features/dashboard/dashboard.component.html
~~~~

Expectations:

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate the usage of `trackBy`.
3. **Evidence:** Quote an `*ngFor` loop that is missing a `trackBy` function.
4. **Suggestion:** If `trackBy` is missing, provide a sample `trackBy` function for the `dashboard.component.ts` class and show how to add it to the `*ngFor` expression.

~~~~
#### **7. Prompt: TypeScript Type Safety**
```markdown
**Goal:**
Audit the project's models and `tsconfig.json` to ensure strict type
safety and avoidance of the `any` type.

**Context:**
You are a type safety agent. The `any` type disables TypeScript's static
type checking. The project should have `strict: true` enabled in
`tsconfig.json`, and all models should be strongly typed using interfaces.
~~~~

~~~~
**Source Code to Analyze:**
```typescript

// src/app/core/models/models.ts

// ALSO, Check tsconfig.json FILE.
~~~~

Expectations:

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate the overall type safety.
3. **Evidence:** Provide an example of `any` being used or point out if `strict` mode is disabled in `tsconfig.json`.
4. **Suggestion:** Recommend replacing `any` with a specific `interface` or `type`.

~~~~
#### **8. Prompt: Code Structure & DRY**
```markdown
**Goal:**
Analyze the project's folder structure and identify any duplicated code
between the `dashboard` and `payment` features.

**Context:**
You are a code organization agent. A well-structured project groups files
by feature. Duplicated logic (e.g., the same user data formatting in two
different components) should be extracted into a reusable service or pipe
in the `shared` folder.

**Source Code to Analyze:**
```typescript

// src/app/features/dashboard/dashboard.component.ts
// AND
// src/app/features/payment/payment.component.ts
~~~~

Expectations:

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate adherence to DRY principles.
3. **Evidence:** Provide examples of duplicated code blocks found in both files.
4. **Suggestion:** Recommend extracting the duplicated logic into a shared service or pipe.

9. Prompt: Function and Line Limits
~~~markdown
**Goal:**
Review the `payment.component.ts` file for functions that are excessively long, indicating they may have too many responsibilities.

**Context:**
You are a code readability agent. For maintainability, functions should be concise and focused (e.g., under 75 lines). Large functions should be broken down into smaller, private helper methods. A payment component often has complex logic that is a good candidate for this review.

**Source Code to Analyze:**
```typescript

// src/app/features/payment/payment.component.ts

**Expectations:**

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate the conciseness of the code.
3. **Evidence:** Identify a function (e.g., a form submission handler) that is overly long and provide its line count.
4. **Suggestion:** Recommend specific logic to be extracted from the long function into smaller, well-named private methods.

#### **10. Prompt: Dependency Injection Scope**
```markdown
**Goal:**
Verify that services are provided at the correct scope, focusing on the `auth` feature.

**Context:**
You are a dependency injection agent. In modern Angular, services intended to be application-wide singletons should be decorated with `@Injectable({ providedIn: 'root' })`. This is tree-shakable and preferred over adding services to a module's `providers` array.

**Source Code to Analyze:**
```typescript
~~~

```
// src/app/features/auth/auth.service.ts
```

Expectations:

1. **Verdict:** `Pass` or `Fail`.
2. **Score (1-10):** Rate the correctness of the service provider scoping.
3. **Evidence:** Quote the `@Injectable` decorator from the service class.
4. **Suggestion:** If the service is a singleton and not using `providedIn: 'root'`, recommend changing it.

**10. Prompt: Agent mode Changes **

```
can you do this changes and rerun the above prompt and check the scores

<!-- end list -->
```