```rust
let rust_notes = r#"=== Rust Quick Reference ===

--- Basics ---
- Entry point: `fn main() {}`.
- Variables are immutable by default: `let x = 5;`.
- Make mutable with `mut`: `let mut y = 10;`.
- Constants: `const MAX: u32 = 100;` (must have a type, evaluated at compile time).
- Primitive types: `bool`, `char`, integers (`u8`…`u128`, `i8`…`i128`), floating (`f32`, `f64`), tuples, arrays, slices.

--- Control Flow ---
- `if`, `else if`, `else`.
- `match` for pattern matching (exhaustive).
- Loops: `loop {}`, `while condition {}`, `for item in iter {}`.
- Loop labels and `break`/`continue` with values.

--- Functions ---
- Declaration: `fn name(arg1: Type1, arg2: Type2) -> ReturnType { … }`.
- Single-expression bodies: `fn add(a: i32, b: i32) -> i32 { a + b }`.
- No implicit conversion – all types must match exactly.
- Default arguments are not supported; use optional parameters (`Option<T>`) or overload with traits.

--- Ownership ---
- Each value has a single owner.
- When the owner goes out of scope, the value is dropped.
- Move semantics: `let b = a; // a is moved, cannot be used`.
```

- Clone explicitly: `let b = a.clone();` (deep copy if type implements `Clone`).

--- Borrowing ---
- References: `&T` (immutable) and `&mut T` (mutable).
- Rules:
  * Any number of immutable refs **or** exactly one mutable ref at a time.
  * References must not outlive the data they point to.
- Slicing: `&array[0..3]` yields a slice `&[T]`.

--- Lifetimes ---
- Annotate when the compiler cannot infer how long a reference lives.
- Syntax: `fn foo<'a>(x: &'a str) -> &'a str { … }`.
- Lifetime elision rules cover most simple cases.

--- Structs & Enums ---
- Struct: `struct Point { x: f64, y: f64 }`.
- Tuple struct: `struct Color(u8, u8, u8);`.
- Enum with variants:
  ```rust
  enum Message {
      Quit,
      Move { x: i32, y: i32 },
      Write(String),
      ChangeColor(i32, i32, i32),
  }
  ```
- Pattern match on enums via `match`.

--- Traits ---
- Define shared behavior:
  ```rust
  trait Drawable {
      fn draw(&self);
  }
  ```
- Implement for a type: `impl Drawable for Circle { … }`.
- Trait bounds on generics: `fn render<T: Drawable>(item: T) { … }`.
- `dyn Trait` for dynamic dispatch (`Box<dyn Drawable>`).

--- Generics ---
- Type parameters: `fn max<T: PartialOrd>(a: T, b: T) -> T { … }`.
- `where` clause for readability:
  ```rust
  fn foo<T, U>(t: T, u: U)
  where
      T: Clone,
      U: Into<String>,
  { … }
  ```

--- Error Handling ---
- Recoverable errors: `Result<T, E>`.
  ```rust
  fn read_file(path: &str) -> Result<String, std::io::Error> { … }
  let content = read_file("data.txt")?;
  ```

- Unrecoverable errors: `panic!()`.
- `Option<T>` for optional values (`Some(v)` / `None`).

--- Modules & Crates ---
- File layout:
  ```
  src/
    lib.rs   // crate root for libraries
    main.rs  // crate root for binaries
    foo.rs   // module `foo`
    bar/mod.rs // module `bar`
  ```
- Declare modules: `mod foo;` or `pub mod bar;`.
- Use items: `use crate::foo::MyStruct;`.
- Re-export: `pub use foo::MyStruct;`.

--- Cargo ---
- Create: `cargo new my_proj`.
- Build: `cargo build` / `cargo build --release`.
- Run: `cargo run`.
- Test: `cargo test`.
- Add dependency: edit `Cargo.toml` !' `serde = "1.0"` then `cargo fetch`.
- Workspaces for multi crate repos.

--- Common Collections ---
- `Vec<T>`: growable array.
- `String` (`Vec<u8>` with UTF 8 validation).
- `HashMap<K, V>` (requires `Eq + Hash` on `K`).
- `HashSet<T>` (requires `Eq + Hash`).
- `BTreeMap`, `BTreeSet` (ordered).

--- Concurrency ---
- Threads: `std::thread::spawn(move || { … })`.
- Message passing: `std::sync::mpsc::{channel, Sender, Receiver}`.
- Shared mutable state: `Arc<Mutex<T>>` or `Arc<RwLock<T>>`.
- `async/await` (requires an executor, e.g., `tokio`):
  ```rust
  async fn fetch() -> Result<String, reqwest::Error> { … }
  tokio::main
  async fn main() { let data = fetch().await?; }
  ```

--- Memory Safety Guarantees ---
- No data races at compile time.
- No null pointers (use `Option<T>`).
- No dangling pointers (ownership & lifetimes).
- No buffer overflows (bounds checking on slices/Vec).

--- Useful Macros ---
- `println!`, `format!`, `eprintln!`.
- `vec!`, `hashmap!` (from `maplit` crate), `assert!`, `debug_assert!`.
- `#[derive(Debug, Clone, PartialEq, Eq, Hash)]` for common trait implementations.
- `#[cfg(test)]` and `#[test]` for unit tests.

--- Testing ---
```rust
#[cfg(test)]
```

```
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }

    #[should_panic]
    fn fails() {
        panic!("this test should panic");
    }
}
```

--- Formatting & Linting ---
- `cargo fmt` (uses `rustfmt`).
- `cargo clippy` for extra lint checks.

--- Release Checklist ---
- Run `cargo test --all`.
- Run `cargo fmt -- --check`.
- Run `cargo clippy -- -D warnings`.
- Build with `cargo build --release`.
- Audit dependencies (`cargo audit`).

--- Quick Cheat Sheet ---
Concept	Syntax Example
Variable (immut)	`let x = 5;`
Variable (mut)	`let mut y = 10;`

Function	`fn add(a: i32, b: i32) -> i32 { a+b }`
Struct	`struct Point { x: f64, y: f64 }`
Enum	`enum Opt { Some(i32), None }`
Trait	`trait ToString { fn to_str(&self) -> String; }`
Impl	`impl ToString for i32 { … }`
Generic	`fn max<T: PartialOrd>(a: T, b: T) -> T { … }`
Result	`Result<T, E>`
Option	`Option<T>`
Borrow (`&`)	`fn len(s: &str) -> usize { s.len() }`
Mutable borrow	`fn inc(x: &mut i32) { *x += 1; }`
Lifetime	`fn foo<'a>(x: &'a str) -> &'a str { x }`
Module	`mod utils;`
Use	`use crate::utils::helper;`
Cargo add crate	`cargo add serde`
Async fn	`async fn fetch() -> Result<…, …> { … }`
Spawn thread	`std::thread::spawn(move || { … })`

--- End of Notes ---
"#;
```
```