functional Rust→eBPF Aya pipeline (bpfel-unknown-none):

---

1. BPF Linker

- Install bpf-linker (cargo install bpf-linker) on your dev container.
- Configure Cargo to use bpf-linker for the BPF target (see config below).

2. Cargo & Rustflags

- Ensure nightly Rust toolchain (which you have!).
- Pass all the right flags:
    - -Z build-std=core,alloc for cargo build (no std!)
    - --target bpfel-unknown-none (explicit)
    - Rustflags:
        - -C linker=bpf-linker
        - -C panic=abort
        - -C lto=yes
        - -C codegen-units=1
        - (optionally) -C debuginfo=2 -C link-arg=--btf for BTF support

Cargo Config Example:

In kernel/aya/lock-ebpf/.cargo/config.toml (preferred: per eBPF crate!):

```
[build]
target = "bpfel-unknown-none"

[target.bpfel-unknown-none]
linker = "bpf-linker"
rustflags = [
  "-C", "panic=abort",
  "-C", "lto=yes",
  "-C", "codegen-units=1",
  "-C", "debuginfo=2",
  "-C", "link-arg=--btf"
]
```

Do NOT put this at the workspace root or all crates will try to build for BPF.
You want only the eBPF program to see these settings.

---

3. Build Command (as in xtask or script):

Your xtask or build script should call:

```
cargo +nightly build --target bpfel-unknown-none -Z build-std=core,alloc --release
```

- Use --release for optimized code (eBPF code size matters).
- This will invoke bpf-linker if your config is correct.

---

4. Kernel Requirements:

- Minimum: Linux 4.18 (BTF support starts here).
- Best experience: 5.2+ (new BPF features, more stable CO-RE/fentry).

---

5. Verification:

- Check output with file target/bpfel-unknown-none/release/lock — must say “ELF 64-bit LSB relocatable, eBPF, version 1”.
- (Optional) Use llvm-objdump -S to see BPF instructions.
- Try loading with bpftool prog load ... if you want to be certain.

---

6. Custom Script / xtask Tips:

- The script should:
  - Set up the working directory so that relative paths are correct.
  - Clean output first if needed (cargo clean).

    - Build with above flags.
    - Copy the output ELF where you want (xtask usually does this!).
    - Never force the whole workspace to build for BPF.
- Example snippet for an xtask:

```
let status = std::process::Command::new("cargo")
  .args([
    "+nightly", "build",
    "--target", "bpfel-unknown-none",
    "-Z", "build-std=core,alloc",
    "--release"
  ])
  .current_dir("lock-ebpf")
  .status()?;
```

---

7. Final Checklist

- Only your eBPF crate has no_std, special target, and linker settings.
- bpf-linker is installed and in PATH in the container.
- Your user-space loader is built for native (x86_64-unknown-linux-gnu) — never for bpfel-unknown-none.
- You build your eBPF binary with the full set of flags above.
- Your build output is a real ELF eBPF file (see above for verification).

Summary:
You need a working per-crate Cargo config for the eBPF program (pointing to bpf-linker, not at the workspace root), correct build flags in xtask, and a kernel new enough for BTF and your required eBPF features.
The xtask/script just needs to invoke cargo with those flags and handle output ELF file copying