Rules File Comparison Report

Qiushi Liang, Zhiping Zhang

February 27, 2026

Feature tested: Issue #4 — Secure `.py` File Upload
Branches: `test/with-rules` vs `test/no-rules`

1 Code Quality and Consistency with Project Patterns

1.1 Architecture — Request Flow

The `.cursorrules` file specifies a strict layered architecture: `routes/ → controllers/ → services/ → Supabase DB`.

With Rules — follows the prescribed pattern exactly, including a centralized types file and auth middleware:

```
backend/src/
  types/index.ts            <- Centralized type definitions
  lib/supabase.ts           <- Supabase client
  middleware/auth.ts        <- JWT authentication
  routes/files.ts           <- Route definitions
  controllers/fileController.ts
  services/fileService.ts   <- Validation + Storage + DB write
```

Without Rules — similar layering but with a different structure. No auth middleware, no centralized types, no database write:

```
backend/src/
  config/supabase.ts        <- Supabase client (different folder name)
  middleware/upload.ts       <- Multer config (no auth)
  middleware/errorHandler.ts <- Multer error handler
  routes/fileRoutes.ts      <- Route definitions (different file name)
  controllers/fileController.ts
  services/fileService.ts   <- Storage only, no DB write
  app.ts                    <- Separate Express app file
```

Key difference: The with-rules version includes a complete data pipeline (validate → upload to Storage → write to DB → return mapped response). The no-rules version stops at Storage upload and never persists file metadata to the database.

1.2 Authentication

The `.cursorrules` requires JWT for all protected routes and specifies: *"Always include a test case for unauthenticated requests for every protected endpoint."*

With Rules — complete JWT middleware:

```
// backend/src/middleware/auth.ts (WITH rules)
```

```
export function authenticate(req, res, next): void {
  const authHeader = req.headers.authorization;
  if (!authHeader || !authHeader.startsWith(’Bearer ’)) {
    res.status(401).json({ error: ’Missing or invalid authorization header’ });
    return;
  }
  const token = authHeader.split(’ ’)[1];
  const secret = process.env.JWT_SECRET ?? ’’;
  try {
    const decoded = jwt.verify(token, secret) as JwtPayload;
    req.user = decoded;
    next();
  } catch {
    res.status(401).json({ error: ’Invalid or expired token’ });
  }
}
```

Without Rules — no auth middleware at all. The controller hard-codes a fallback user:

```
// backend/src/controllers/fileController.ts (WITHOUT rules)
// TODO(#7): Extract userId from JWT token via auth middleware
const userId = (req as Request & { userId?: string }).userId ?? ’anonymous’;
```

1.3 Type Safety

With Rules — centralized type definitions in `types/index.ts` with interfaces for `JwtPayload`, `AuthenticatedReque`
`FileRecord`, and `FileUploadResponse`.

Without Rules — no centralized type file. Types are scattered across individual files, and `Request` is cast inline with ad-hoc type extensions.

1.4 Database Interaction and Rollback

With Rules — writes a record to the `files` table after uploading, and rolls back the Storage upload if the DB insert fails:

```
// backend/src/services/fileService.ts (WITH rules)
const { data, error: dbError } = await supabase
  .from(’files’)
  .insert({ id: uniqueId, user_id: userId, file_name: originalName,
            storage_path: storagePath, size_bytes: buffer.length })
  .select().single();

if (dbError) {
  await supabase.storage.from(STORAGE_BUCKET).remove([storagePath]);
  throw new Error(‘Database insert failed: ${dbError.message}‘);
}
```

Without Rules — only uploads to Storage. No database persistence, no rollback logic.

1.5 File Validation — Empty File Handling

With Rules — validates against empty (0-byte) files with a dedicated check. **Without Rules** — no empty file check exists anywhere in the codebase.

2 Design/Mockup Intent

The `.cursorrules` references `project-memory/mockup.jpg` and specifies a layout with a file sidebar on the left, a code editor in the center, and an instructions panel on the right. The mockup uses a dark theme with purple accent colors.

2.1 Overall Layout

With Rules — three-section layout matching the mockup (header with nav tabs + sidebar + main area):

```
<!-- frontend/src/App.tsx (WITH rules) -->
<header> InstructScan | [Editor] [History] [Settings] </header>
<div class="flex">
  <FileUploader />        <!-- Left sidebar with file list -->
  <main class="flex-1">   <!-- Center editor area -->
</div>
```

Without Rules — simple centered layout, no sidebar, no navigation tabs:

```
<!-- frontend/src/App.tsx (WITHOUT rules) -->
<header> InstructScan </header>
<main class="flex items-center justify-center">
  <FileUploader />   <!-- Centered upload area only -->
</main>
```

2.2 File List Sidebar

With Rules — dedicated file list matching the mockup: a `FILES` heading, scrollable list of uploaded filenames, selected file highlighted with `border-purple-500`.

Without Rules — no file list at all. The component is a standalone upload area with no concept of browsing previously uploaded files.

2.3 Brand Color

Element	Mockup	With Rules	Without Rules
Logo	Purple	`purple-400`	`indigo-400`
Active tab	Purple	`purple-600`	(no tabs)
Selected file	Purple	`purple-500`	(no file list)
Progress bar	—	`purple-500`	`indigo-500`

3 Adherence to Naming Conventions and Architecture

3.1 File and Route Naming

3.2 API Response Field Naming

The `.cursorrules` states: *"API response bodies must use camelCase. Map database snake_case fields to camelCase equivalents at the controller layer."*

With Rules: `{"fileName": "main.py", "sizeBytes": 1024, "uploadedAt": "..."}` — all camelCase.

Item	With Rules	Without Rules	Convention
Route file	`routes/files.ts`	`routes/fileRoutes.ts`	Folder already implies "routes"
Supabase client	`lib/supabase.ts`	`config/supabase.ts`	Rules specify `lib/`
Type definitions	`types/index.ts`	(none)	Rules require centralized types
API path	`/api/files`	`/files/upload`	Rules require `/api` prefix

Without Rules: {`"filename": "main.py", "storagePath": "...", "sizeBytes": 1024`} — `filename` is all lowercase, breaking the convention.

3.3 Frontend Component Architecture

With Rules — upload logic extracted into a reusable hook (`hooks/useFileUpload.ts`), with the `FileUploader` component receiving state via props. Three files with clear separation: component, hook, API utility.

Without Rules — all logic embedded directly in the component (287 lines of mixed UI and state). Only two files: component and API utility.

3.4 Commit Message Format

`.cursorrules` specifies: `type(scope): description #issueNumber`.

With Rules: `feat(upload): implement secure .py file upload with rules #4` — has scope, references issue.

Without Rules: `feat: implement .py file upload without rules` — missing scope, no issue reference.

4 Quality of Tests Generated

4.1 Test Count Summary

Area	With Rules	Without Rules
Backend — file validation	8	7
Backend — auth middleware	5	0 (no auth)
Backend — controller	0	4
Backend — upload middleware	0	10
Frontend — FileUploader	9	14
Total	**22**	**35**

4.2 Unauthenticated Request Testing

With Rules — 4 dedicated auth failure tests: missing header, wrong scheme (`Basic`), invalid token, expired token. All return 401.

Without Rules — no authentication tests exist because auth was never implemented. The closest test verifies fallback to `'anonymous'` — a significant security gap.

4.3 Frontend Test Approach

Both test suites are competent. The with-rules tests verify **file list and selection behavior** (unique to the mockup-aware implementation), while the without-rules tests focus on **upload state transitions** in the standalone component.

5 Summary

Dimension	With Rules	Without Rules	Verdict
Architecture	Strict layering, auth, DB write, rollback	Similar but no auth, no DB	With rules
Mockup fidelity	3-panel layout, purple theme, nav tabs	Centered upload, indigo	With rules
Naming	Consistent camelCase, `/api` prefix	`filename`, no prefix	With rules
Tests	22, with auth coverage	35, but no auth tests	Tie
Empty file check	Yes	No	With rules
Error rollback	Yes	No	With rules

Conclusion

The `.cursorrules` file produced a measurably better result across every dimension. Authentication was entirely skipped without rules; the mockup was ignored; naming inconsistencies appeared (`filename` vs `fileName`); and database persistence was missing. The rules file acted as a comprehensive specification that kept the AI aligned with project conventions, security requirements, and design intent that it would otherwise have no way to infer from a brief prompt alone.