Deliverables_First Architecture Blueprint (DFA-B): A Traceable Governance Model

I. Executive Summary

The Deliverables-First Architecture Blueprint (DFA-B) mandates a shift in project execution philosophy from focusing primarily on tasks (Work Breakdown Structure, WBS) to defining verifiable product outputs (Product Breakdown Structure, PBS) and formal Deliverable Definition Sheets (DDS) upfront. This methodology is designed for high-compliance and complex domains, integrating rigorous governance with executable proof artifacts.

The core strategy operationalizes a light V-Model approach by enforcing three mandatory design constraints: **Deliverables-First** (PBS precedes WBS), **Contract-First** (interfaces defined formally via OpenAPI and JSON Schema), and **Acceptance-First** (behavior defined using Gherkin before implementation).[1]

The DFA-B achieves maximum, auditable traceability by linking these artifacts—requirements, architectural decisions (ADRs), contracts, tests, and evidence—through a live, granular Requirements Traceability Matrix (RTM). This structure ensures that architectural governance is not passive prose but an actively enforced mechanism, minimizing architectural drift and providing immediate, executable verification proof required for high-security and regulated environments.

II. Literature Review: Foundations of Verified Architecture

The development of the DFA-B synthesizes several proven industry practices, focusing on formalizing planning, governance, and verification to create a verifiable supply chain for software artifacts.

A. Architectural Decision Formalization: The MADR Standard

Architectural decisions (ADs) represent justified design choices that resolve functional or non-functional requirements of architectural significance.[2] Documenting these choices is critical for long-term project viability, knowledge transfer, and auditing. The Markdown Architectural Decision Record (MADR) format provides a streamlined, lean template for capturing these records.[2] This template prioritizes structure, ease of version control, and consumption, aligning perfectly with standard software development repository practices.[2]

Tooling support, such as adr-tools, standardizes the file naming convention (NNNN-title-with-dashes.md), which promotes consistency and eases automated indexing of governance artifacts.[2] This approach transforms architecture documentation from static prose into indexed, version-controlled assets.

The rigor of MADR adoption is often hampered by common documentation failures, referred to as anti-patterns.[2] One prevalent failure is the "Fairy Tale" or "Sales Pitch," where justifications are shallow, risks are ignored, or overly enthusiastic marketing language replaces factual analysis.[2] This failure directly introduces architectural drift and technical debt because the long-term, non-trivial consequences of a decision are not documented or accounted for. For instance, the benefit of "decoupled communication" provided by an event-based architecture may be praised, while the associated "extra design, implementation, and test effort" is omitted, embodying the "Free Lunch Coupon" anti-pattern.[2] To counter this, the DFA-B requires ADRs to explicitly link choices back to the high-priority Non-Functional Requirements (NFRs) defined in the planning phase, ensuring that choices are balanced and risks are acknowledged.

Another critical anti-pattern is the "Mega-ADR" or "Blueprint in Disguise," where extensive, multi-page design details are crammed into the decision record.[2] This practice confuses architectural governance (the *why*) with implementation documentation (the *how*). The DFA-B methodology insists that ADRs focus exclusively on the context, the decision rationale, and the consequences. Detailed component interactions and implementation specifics must reside in separate technical documents or code, linked from the ADR, thereby maintaining the record's primary focus on justified choice. Furthermore, avoiding "Tunnel Vision"—which occurs when consequences for essential stakeholders like operations and maintenance staff are ignored—is enforced by mandating that every ADR be assessed against critical NFRs like Operability and Evoluability.[2]

B. Product-Based Planning and WBS Derivation (PBS)

Effective project management relies on the hierarchical decomposition of scope, traditionally defined as the Work Breakdown Structure (WBS).[4] However, for high-rigor projects, the foundation must be the Product Breakdown Structure (PBS), which organizes the scope entirely around verifiable, output-oriented components.[5] This methodology adheres to the foundational design principle: **plan outcomes, not actions**.[4]

Deliverables represent the tangible (e.g., software) or intangible (e.g., training program) outputs a project must produce.[6] They serve as critical benchmarks for both progress and quality.[6] In DFA, process deliverables (like status reports) are distinguished from product deliverables (the final outcomes).[7]

The formal artifact mandated for this phase is the Deliverable Definition Sheet (DDS).[8] The DDS acts as the governing document for each PBS terminal element, defining its description, stakeholder, expected outcome, and, crucially, the verification method.[9] The DDS ID becomes the unique traceability anchor in the RTM. By requiring the PBS and DDS to be finalized first, the subsequent WBS can only be derived from the tasks necessary to produce the DDS-defined outputs. This mandated sequence ensures adherence to the "100% rule" (all work contributes to defined outputs) and systematically prevents the introduction of tasks (and associated effort) not tied to a measurable, verified deliverable.[4]

C. Acceptance-First Development: ATDD/BDD

Behavior-Driven Development (BDD), facilitated by tools like Cucumber using Gherkin syntax, allows teams to define flow and expectations using human-readable language.[10] This practice transforms the acceptance criteria (ACs) into a living requirements document that non-technical stakeholders, such as product managers, can read, approve, and collaborate on.[11]

The core technical strength of BDD integration lies in the abstraction principle: the imperative implementation details (the code that calls the API, manipulates the data) reside in the step definition layer.[12] Consequently, if the underlying implementation changes, only the step definitions require modification, leaving the business-focused feature files untouched.[12] For DFA-B, this principle is crucial, as Gherkin scenarios must always be written in business language, avoiding references to technical concepts or UI components.[10]

Furthermore, acceptance testing often implicitly covers integration points. A single scenario defining an overarching behavior (e.g., "Secure Customer Onboarding") typically necessitates calling multiple underlying API methods within the step definitions (e.g., first a POST, then a GET for verification).[12] By mandating Acceptance-First (writing Gherkin before implementation), the DFA-B establishes the **Micro-verification** layer of the light V-Model early in the project lifecycle.[1]

D. Interface Rigor: Contract-First Design

The Contract-First mandate requires that every service interface be defined by a formal, language-agnostic contract. For REST services, this typically involves OpenAPI Specification (OAS); for payloads and data models, JSON Schema is used.[13] JSON Schema is a standard for defining and validating the structure, constraints, and data types of JSON documents.[13]

In a microservices environment, contract testing using JSON Schema is essential for guaranteeing that the data exchanged between services is consistent and adheres to expectations.[14] This practice is critical for reducing tight coupling; services only rely on the contract, not the internal implementation details of the producer.[14] For Node.js/TypeScript environments, validation libraries like Ajv are used to compile and enforce the JSON Schema contract at runtime within the test suite, such as Jest.[14]

The synthesis of Contract-First and Acceptance-First methodologies is powerful. The DFA-B approach dictates that the contract must be defined immediately after the architectural decision (ADR) on the communication style. This contract then becomes the technical specification that the Gherkin step definitions *use* and *verify*.[12] While compiled languages benefit from code generation to enforce contracts at compile time, the TypeScript stack relies on running dedicated Contract Tests (using Ajv/Jest) alongside the BDD tests to ensure that request and response payloads adhere perfectly to the formal contract definitions. This dual testing strategy provides necessary executable proof of contract enforcement, mitigating runtime errors caused by contract violations.[14]

E. Verification and Validation Integration: The Light V-Model

The V-Model provides a structured and controlled approach to development, particularly valuable in regulated fields requiring high safety or security, where stable requirements are prioritized.[1] The Light V-Model adapts this structure to Agile environments by integrating

verification and validation iteratively.

The core principle involves planning verification concurrently with planning the artifact itself. In the Agile V-Model, verification is segmented: **Micro-level verification** is achieved through the Acceptance Criteria of individual user stories (our Gherkin scenarios), while **Macro-level verification** focuses on the overall Definition of Done and cross-system standards.[1] Validation, conversely, focuses on stakeholder or user acceptance testing (UAT).

DFA-B formalizes the V-Model structure by using the Acceptance-First mandate to define Micro-level verification early on (the Gherkin features). The RTM then serves as the closure mechanism for the V-Model, demonstrating proof that every Deliverable (defined at the high-level planning phase) is successfully matched and verified by executable tests (defined at the low-level testing phase).[16]

F. End-to-End Traceability and Executable Proof

Traceability is the explicit linkage between development artifacts, such as requirements, designs, code, and tests.[17] The Requirements Traceability Matrix (RTM) is the strategic document that maps requirements to their satisfying deliverables, ensuring completeness and visibility for stakeholders.[18] In Agile contexts, this RTM must be a living document that adapts to iterative development, linking user stories (or DDS IDs) to epics, acceptance criteria, and test cases.[19]

The DFA-B methodology extends this concept to achieve **executable traceability**. This means the RTM must link directly to automated, executable test cases, providing bidirectional links down to the actual source code that implements the feature.[16] To achieve this unprecedented granularity, the DFA-B introduces the mandatory **File-level Definition of Functionality and Deliverables (DoFD)**. This structural requirement dictates that metadata—including the associated Deliverable ID and the path to the verification test—must be embedded directly within the file header of the source code.[17] A continuous integration (CI) script can then parse these headers, extract the necessary paths, and link them to the test execution evidence (CI log URLs), thereby generating a truly auditable and executable RTM.[16]

G. Dependency Mapping and Visualization

Architectural governance requires enforcing technical boundaries and detecting systemic issues like circular dependencies. Dependency graphs are instrumental in this, summarizing package relationships and providing a clear overview of the project structure to identify bottlenecks and aid refactoring efforts.[20]

For the TypeScript environment, tools like dependency-cruiser and ts-dependency-graph can analyze the codebase and output the dependency structure.[22] A critical feature for integration into modern documentation and CI pipelines is the ability to output diagrams using Mermaid syntax.[24] Mermaid is widely adopted, supported natively by platforms like GitHub, allowing the dependency map to be checked and displayed directly as a text artifact within the repository.[24] By configuring dependency-cruiser with rules, the architectural boundaries defined in the ADRs (e.g., Layering constraints) are transformed into technical rules that the CI actively enforces.[23]

III. Comparison Matrix: Methodology Integration Points

The Deliverables-First Architecture Blueprint (DFA-B) mandates a fundamental operational shift compared to traditional, scope-driven development, specifically by front-loading verification and accountability into the planning artifacts.

Table 1: Comparison of Traditional vs. Deliverables-First Architecture (DFA)

Dimension	**Traditional Scope-Driven (WBS-First)**	**Deliverables-First Architecture (DFA)**
Planning Driver	Tasks/Effort (WBS)	Product/Output (PBS, DDS) [4, 6]
Verification Timing (V-Model)	End-of-phase or post-implementation testing (Sequential)	Acceptance-First (Pre-implementation) defining Micro-Verification (AC) [1]
Interface Source of Truth	Implementation-driven, Documentation-last	Contract-First (OAS/Schema as enforceable artifact) [12,

		14]
Architectural Governance	Static documentation (e.g., Wiki, long SAD)	MADR enforced via artifacts (Contract, Dependency rules checked by CI) [2]
Traceability Granularity	Requirements $\rightarrow$ Test Case (Coarse/Manual)	RTM: Deliverable $\leftrightarrow$ Test $\leftrightarrow$ Evidence $\leftrightarrow$ File (Auditable/Automated) [16]

The DFA-B is distinguished by its mandate to push verification upstream. By adopting Acceptance-First and Contract-First, the methodology integrates verification activities directly into the design and planning phases, aligning with the V-Model's core directive to define how an artifact will be verified before implementation commences.[1] The reliance on executable artifacts—OpenAPI, Gherkin features, DoFD headers—means that architectural decisions are not passively reviewed documents, but actively tested constraints enforced by the continuous integration pipeline. This transition from passive document review to active, executable governance ensures compliance is built-in and auditable in real-time.

IV. Deliverables-First Architecture Blueprint (DFA-B)

The DFA-B defines a structured, mandatory sequence for artifact creation, ensuring that every output is traceable from the initial high-level requirement to the final line of code and its passing test evidence.

A. DFA Process Flow: A Traceability-First Loop

The process mandates an artifact creation order that prioritizes verification planning and architectural governance before implementation begins.

Code snippet

```
graph TD
    A --> B{1. Build PBS/DDS};
    B --> C;
    C --> D;
    D --> E;
    E --> F;
    F --> G;
    G --> H[7. Implementation/Code];
    H --> I;
    I --> J;
    J --> K{10. Evidence (Audit/Compliance)};
    K --> L;

    style J fill:#f9f,stroke:#333,stroke-width:2px
    style G fill:#ccf,stroke:#333,stroke-width:2px
```

The flow is engineered to establish traceability early. Step 6, creating RTM links, is executed prior to implementation, formalizing the V-Model closure where the planning elements (DDS, Contract) are explicitly mapped to the verification elements (Gherkin). Step 9, CI Execution, serves as the engine of executable governance, transforming the RTM structure into a dynamically verifiable report.[16]

B. Planning Artifacts: DDS and PBS $\rightarrow$ WBS

The Product Breakdown Structure (PBS) must fully define the project scope in terms of deliverables.[4] Each terminal element of the PBS must be formalized using a Deliverable Definition Sheet (DDS).[8]

The DDS is the primary planning artifact that captures the expected outcome, the NFRs addressed, and the required verification method, providing the initial anchor for the RTM. By defining the deliverables and their acceptance criteria upfront, the subsequent Work Breakdown Structure (WBS) is constrained to only those tasks strictly necessary to achieve the defined outputs. This ensures that resources are allocated solely to verifiable outcomes, eliminating "orphan tasks" that do not contribute to a defined project output.

C. Decision Management: Formalizing ADRs and Anti-Patterns

ADRs must adhere to the MADR template, detailing the Context, Decision, Status, and Consequences.[2] Within the DFA-B, the ADR phase directly precedes contract definition, ensuring that the choice of interface style (e.g., REST/OpenAPI versus gRPC/Proto) is a governed decision, explicitly linking NFRs (such as security or latency) to the resulting technical standard.

Governance during the ADR phase relies on actively checking against known anti-patterns.[2] For example, reviewers must explicitly demand consequences that address the long-term maintainability and operational burdens—countering the "Free Lunch Coupon" anti-pattern.[2] Furthermore, ADRs must remain high-level, focusing on the *why* of the architecture, rather than descending into detailed design specifications, thereby avoiding the "Mega-ADR" trap.[2] The ADR must clearly state how the chosen solution scores against the prioritized NFRs (e.g., Operability and Security).

D. Traceability Framework: RTM Design and Automation

The RTM is the core accountability artifact, providing proof of compliance and ensuring that requirements are fully met.[19] The DFA RTM is designed for maximum granularity and automated verification.

Table 2: Deliverables-First RTM Schema

Field	Description	Source Artifact	Traceability Type
DDS ID	Unique identifier for the Deliverable (PBS element)	DDS/PBS	Backward (Requirement origin)
Contract Spec Path	File path to the mandatory interface contract (e.g., openapi/customer.yaml)	OpenAPI/JSON Schema	Bidirectional (Design definition)

Gherkin Feature Path	File path to the Acceptance Criteria (Feature/Scenario)	Gherkin Feature File	Forward (Verification plan)
Test/Step Definition Path	File path to the executable Cucumber.js step file	TypeScript Code	Forward (Execution logic)
File DoFD Path(s)	Path(s) to the implementing source file(s)	Source Code (via DoFD header)	Bidirectional (Implementation evidence)
CI Evidence URL	URL to the passing test run log in CI (Executable proof)	GitHub Actions/CI Log	Forward (Verification result)

The automation strategy mandates that the RTM is generated dynamically during the CI pipeline execution. A dedicated script parses structured data from the DDS inventory, Gherkin features, and critically, scrapes the DoFD metadata embedded in source files to establish the implementation link. This ensures that the RTM links resolve to real, executable file paths and passing CI evidence, transforming the matrix into an active audit trail.[16]

V. End-to-End Exemplar: Secure Customer Onboarding API (TypeScript/Node 20)

This exemplar demonstrates the operationalization of the DFA-B methodology for a high-compliance use case in a FinTech context.

Project Parameters:

- **Domain/Context:** Highly Regulated FinTech/Payment Processing Gateway.
- **Feature/Use-Case:** Securely Onboarding a New Customer via REST API (POST /customers).
- **Target Stack:** TypeScript/Node 20; Test Stack: Cucumber.js + Jest.
- **Constraints:** PCI-DSS Compliance (data tokenization, audit trail), Latency < 150ms.
- **NFRs:** Security (A-Priority), Operability (A-Priority), Evoluability (B-Priority).

A. Planning Artifacts: PBS and DDS

The decomposition focuses strictly on product outputs.

PBS Structure:

```
1. Customer Onboarding Capability
   1.1. Customer API Service
      1.1.1. Secure Customer Creation Endpoint
      1.1.2. Customer Data Persistence Layer
   1.2. Audit and Compliance Logging Module
```

DDS Example (DDS-001): Secure Customer Creation Endpoint

```
Code snippet
```

```
Deliverable Definition Sheet (DDS-001)

| Field | Value |
|---|---|
| Deliverable ID | DDS-001 |
| Name | Secure Customer Creation Endpoint (POST /customers) |
| Description | REST endpoint accepting validated customer data, tokenizing sensitive fields, and persisting the resulting entity ID. Must ensure PII and payment data is handled according to PCI-DSS standards. |
| Owner | API Team Lead |
| Dependencies (PBS IDs) | DDS-002 (Persistence Layer), 1.2 (Audit Module) |
| NFRs Addressed | Security (PCI-DSS), Latency (<150ms) |
| Verification Method | Acceptance Testing (Gherkin feature path: features/onboarding.feature) |
| Evidence Path | RTM link to CI-generated proof log (e.g.,.github/workflows/ci-build/onboarding-test-result.log). |
```

B. Architectural Decision (ADR 0001)

This ADR governs the mandatory technical standard for interface definition and verification, directly addressing the Security NFRs.

Code snippet

Architecture Decision Record 0001: Contract Standardization (OpenAPI + JSON Schema)

1. Context
The FinTech domain requires extreme rigor in data handling due to PCI-DSS and audit requirements. We need robust, automated validation of all inbound and outbound payloads. Given our TypeScript/Node.js stack, relying solely on compile-time checks is insufficient for external contract adherence. We must ensure schema validity at runtime and have an explicit contract artifact that serves as the source of truth.

2. Decision
We adopt a Contract-First approach using **OpenAPI Specification (OAS 3.0)** for resource definition and **JSON Schema Draft 7** for detailed request/response payload validation. Contract enforcement will be performed immediately upon request receipt and validation execution will be integrated into the Cucumber step definitions using the **Ajv** library. This guarantees validation occurs during the Micro-verification testing phase.

3. Status
Accepted

4. Consequences
(+) Payload structure and data type enforcement are automated and auditable, critical for PCI-DSS compliance.
(+) JSON Schema reduces tight coupling between services by providing a single, reliable definition.
(+) The use of Ajv in the request pipeline and in testing allows "fail fast" contract violation detection.
(-) Requires strict developer discipline to maintain the contract artifacts (OAS/Schema) as the definitive source of truth, avoiding the "Blueprint in Disguise" anti-pattern.
(-) Adds necessary runtime overhead for schema validation, which must be accounted for in

the Latency constraint.

C. Contract-First Artifacts

The contracts define the API structure and constraints, serving as the technical specification for DDS-001.

- **OpenAPI Stub (openapi/onboarding.yaml):**

YAML

```yaml
# openapi/onboarding.yaml
openapi: 3.0.0
info:
  title: Customer Onboarding API
  version: 1.0.0
paths:
  /customers:
    post:
      summary: Creates a new customer record securely.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CustomerCreateRequest'
      responses:
        '201':
          description: Customer created successfully
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/CustomerCreateResponse'
components:
  schemas:
    CustomerCreateRequest:
      $ref: './schemas/CustomerCreateRequest.json'
```

```
  CustomerCreateResponse:
    $ref: './schemas/CustomerCreateResponse.json'
```

- **JSON Schema Artifact (openapi/schemas/CustomerCreateRequest.json):** This schema enforces structural integrity and defines expected data formats.

JSON

```
// openapi/schemas/CustomerCreateRequest.json
{
  "$id": "CustomerCreateRequest.json",
  "title": "Customer Creation Payload",
  "type": "object",
  "properties": {
    "name": { "type": "string", "description": "Customer Full Name" },
    "email": { "type": "string", "format": "email" },
    "paymentDetails": {
      "type": "object",
      "properties": {
        "cardNumber": { "type": "string", "minLength": 15, "maxLength": 19 }
      },
      "required": ["cardNumber"]
    }
  },
  "required":,
  "additionalProperties": false
}
```

D. Acceptance-First Artifacts (Gherkin Feature File)

The Gherkin scenario defines the verifiable behavior for DDS-001, written in business language.

Gherkin

```
# links to DDS-001
Feature: Secure Customer Onboarding
  As a Payment Gateway Operator,
  I want to securely onboard new customers via API,
  So that I can process transactions while maintaining PCI-DSS compliance.

  Scenario: Successful customer creation with valid, tokenized payment details
    Given the Onboarding API is running
    And I have a valid CustomerCreateRequest payload conforming to the contract
    When I send a POST request to "/customers" with the payload
    Then the response status should be 201 Created
    And the response body should conform to the CustomerCreateResponse contract
    And the database should contain a new Customer record linked to a tokenized payment ID
```

E. Implementation and File-level DoFD

The source code includes the DoFD header, providing the final, executable link for traceability.[17] This header allows automated RTM generators to confirm which deliverable is implemented by this file and where the evidence resides.

- **Example Implementation File (src/api/customer_api.ts):**

TypeScript

```
/**
 * File-Level Definition of Functionality and Deliverables (DoFD)
 *
 * @Deliverable DDS-001: Secure Customer Creation Endpoint
 * @Description Handles validation, tokenization, and delegation to the persistence layer.
 * @Tests features/onboarding.feature::Scenario: Successful customer creation...
 * @EvidencePath./.github/workflows/ci-build/onboarding-test-result.log
 */
import { Router } from 'express';
import Ajv from 'ajv';
import requestSchema from '../../openapi/schemas/CustomerCreateRequest.json';
```

```
// ADR-0001 implementation: Instantiation and use of Ajv for contract enforcement
const ajv = new Ajv({ schemas: });
const validate = ajv.getSchema('CustomerCreateRequest.json');

const router = Router();

router.post('/customers', (req, res) => {
  // 1. Contract Validation (Micro-verification enforcement)
  // Ensures incoming payload adheres to the contract, fulfilling the Security NFR.
  if (!validate ||!validate(req.body)) {
    console.error('Contract validation errors:', validate.errors);
    return res.status(400).json({ error: 'Request failed schema validation.' });
  }

  // 2. Business Logic: Tokenization and Delegation (calling DDS-002)
  //...
  res.status(201).json({ customerId: 'CUST-123', token: 'TOKEN-XYZ' });
});

export default router;
```

F. Executable Traceability Matrix (RTM Artifact)

This dynamically generated artifact demonstrates end-to-end traceability, ensuring every DDS is linked through all phases to executable evidence.

Table 3: Executable RTM Snapshot (Generated Artifact)

DDS ID	Contract Spec Path	Gherkin Feature Path	Test/Step Definition Path	File DoFD Path(s)	CI Evidence URL (Placeholder)	Status
DDS-001	openapi/onboarding.yaml	features/onboarding.feature	tests/steps/api.steps.ts	src/api/customer_api.ts	/ci-results/tests/onboarding-feature	PASS

					.log	
DDS-002	N/A (Internal)	N/A	tests/unit/persistence.test.ts	src/data/customer_repo.ts	/ci-results/tests/persistence.log	PASS

G. Dependency Mapping and Visualization

The dependency graph provides visual governance and is generated and checked automatically in CI using dependency-cruiser configured for Mermaid output.[23]

Code snippet

```
graph TD
  subgraph FinTech Application
    direction LR
    A -- References/Enforces --> F(OpenAPI Contract);
    A --> B(Business Layer);
    B --> C;
    B --> D[Audit Module];
  end

  E -- Verifies --> A;
  E -- Validates Against --> F;

  style F fill:#ADD8E6,stroke:#333,stroke-dasharray: 5 5
  style E fill:#90EE90
```

This graph enforces critical architectural layering, ensuring, for example, that the Persistence layer does not import components from the API layer, thereby upholding the Evoluability NFR defined in the planning phase.

H. Repository Structure and Run Instructions

Repository Structure:

Code snippet

```
.
├──.github/
│   └── workflows/
│       └── ci.yaml
├── docs/
│   ├── adrs/
│   │   └── 0001-contract-standardization.md
│   ├── DDS/
│   │   └── dds-001-customer-creation.md
│   ├── RTM.csv
│   └── dependency_map.md
├── features/
│   └── onboarding.feature
├── openapi/
│   ├── onboarding.yaml
│   └── schemas/
│       ├── CustomerCreateRequest.json
│       └── CustomerCreateResponse.json
├── src/
│   ├── api/
│   │   └── customer_api.ts (DoFD metadata)
│   └── data/
│       └── customer_repo.ts
├── tests/
│   ├── steps/
│   │   └── api.steps.ts (Gherkin step definitions, uses Ajv)
│   └── support/
│       └── world.ts
├── scripts/
│   └── generate_rtm.js
└── package.json
```

CI/CD Pipeline Plan (GitHub Actions):
The pipeline executes tests, validates governance artifacts, and generates the traceability reports.

YAML

```yaml
#.github/workflows/ci.yaml (Mandatory Artifact Generation)
name: DFA Validation Pipeline
on: [push]
jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      # Standard setup steps omitted for brevity

      - name: Contract Validation (OpenAPI/JSON Schema Check)
        # Ensures contracts are structurally valid against the OAS standard
        run: npm run validate:contracts

      - name: BDD and Contract Tests (Micro-Verification)
        # Runs Cucumber/Jest, including Ajv validation within step definitions
        run: npm run test:bdd >.github/workflows/ci-build/onboarding-test-result.log

      - name: Generate Dependency Graph (Architectural Governance)
        # Checks for structural violations against ADR rules
        run: npx dependency-cruiser --output-to mermaid > docs/dependency_map.md

      - name: Generate Live RTM
        # Custom script to parse DDS, Gherkin, DoFD tags, and link to evidence log
        run: node scripts/generate_rtm.js

      - name: Upload Artifacts
        uses: actions/upload-artifact@v4
        with:
          name: DFA_Artifacts
          path: |
            docs/RTM.csv
            docs/dependency_map.md
            .github/workflows/ci-build/
```

How to Run Locally:

1. **Prerequisites:** Node 20+ runtime environment.
2. **Setup:** npm install (Installs necessary dependencies: Cucumber.js, Jest, Ajv, dependency-cruiser).
3. **Validate Contracts:** npm run validate:contracts (Verifies schema integrity).
4. **Run Tests:** npm run test:bdd (Executes BDD scenarios and contract enforcement).
5. **Generate Traceability:**
    - npx dependency-cruiser --output-to mermaid > docs/dependency_map.md (Generates dependency map).
    - node scripts/generate_rtm.js (Generates the final docs/RTM.csv).

VI. Adoption Playbook: Implementation Strategy

The successful adoption of the DFA-B methodology relies on mandatory toolchain integration and strict process governance to sustain the integrity of the traceability links.

A. Toolchain Integration and Configuration

The primary technical effort involves marrying verification tools with the contract artifacts. In the TypeScript environment, the integration of Ajv within the Cucumber step definitions is paramount. The BDD step file (api.steps.ts) must not rely on simple object checks but must execute the validate function against the JSON Schema defined in ADR 0001.[14] This integration ensures that the business-level acceptance test is simultaneously enforcing the technical contract, proving bidirectional quality control.

For architectural governance, dependency-cruiser must be configured with specific rules that map to the module boundaries established in the architecture documentation.[23] The CI pipeline must fail immediately if a circular dependency is detected or if an unauthorized layer boundary is crossed, preventing architectural degradation. The resulting Mermaid output must be a continuously updated artifact in the documentation repository (docs/).[24]

Finally, the RTM automation script (generate_rtm.js) is the critical enabler of executable traceability. This script must programmatically scan the project directory to locate DoFD headers in implementation files.[17] By linking the @Deliverable tag in the code file to the DDS ID entry in the RTM, and referencing the @EvidencePath to the output of the automated tests, the script generates a robust, auditable link from code implementation back to the original

deliverable definition.

B. Process Governance Checklist

Maintaining the integrity of the DFA requires continuous vigilance through formal checkpoints, ensuring the system remains auditable, especially in regulated contexts.[26]

Table 4: DFA Governance and Audit Checklist

Checklist Item	**Phase**	**Artifact Check**	**Traceability Link Verified**
Contract Freeze	Design (Pre-WBS)	ADR 0001, OpenAPI, JSON Schema	DDS $\leftrightarrow$ Contract Spec Path
Acceptance Sign-off	Planning (Pre-Implementation)	Gherkin Feature File	DDS $\leftrightarrow$ Gherkin Feature Path
CI Traceability Run	CI/CD	RTM.csv, CI Log	Evidence Path resolves to CI Log URL
File DoFD Adherence	Implementation	Source File Headers	DoFD Path(s) $\leftrightarrow$ DDS ID
Architectural Boundary	Continuous Integration	Dependency Graph (Mermaid)	ADR-defined module boundaries are enforced by dependency-cruiser [23]

C. Organizational Change Management

The DFA-B methodology necessitates a cultural shift, particularly around collaboration and definition ownership. To mitigate BDD anti-patterns, such as "Tunnel Vision" or the "Copy Edit" focus [2], Gherkin features must be the outcome of collaborative sessions involving Product, Development, and Quality Assurance roles. This ensures the scenarios capture true business behavior rather than implementation details.[10]

Contract ownership is non-negotiable. The service producer must own the OpenAPI/JSON Schema contract, but the organization must establish clear data-sharing agreements to ensure all consuming services use the same contract definition for their own internal contract tests.[27] This disciplined approach prevents undocumented contract deviations and sustains the integrity of the overall microservice ecosystem.

VII. References

1. Architectural Decision Records: MADR Format (Markdown Architectural Decision Records). *adr.github.io/madr/* [2]
2. Olzzio. How to Create Architectural Decision Records (ADRs) — and How Not To (Anti-patterns). *medium.com/olzzio/how-to-create-architectural-decision-records-adrs-and-how-not-to-93b5b4b33080* [2]
3. Project Management Institute (PMI) and Wikipedia. Work Breakdown Structure (WBS) and the 100% Rule. *en.wikipedia.org/wiki/Work_breakdown_structure* [4]
4. Atlassian / Adobe. Principles of Project Deliverables and PBS derivation. *atlassian.com/work-management/project-management/project-deliverables; business.adobe.com/blog/basics/project-deliverables* [6]
5. AIoT Playbook / Knowledge Hut. The Agile V-Model: Micro and Macro Verification in regulated environments. *aiotplaybook.org/index.php?title=Agile_V-Model; knowledgehut.com/blog/agile/v-model-vs-agile* [1]
6. Nakhare, S. Exploring JSON Schema Validation, Contract Testing, and Dynamic Forms. *sohamnakhare.medium.com/exploring-json-schema-validation-contract-testing-and-dynamic-forms-c0472f4de2de* [14]
7. ConnectCD. Test Automation of a Microservice using Cucumber and OpenAPI (Pattern Analysis). *medium.com/connectcd/test-automation-of-a-microservice-using-cucumber-java-and-openapi-c9bac9b7465d* [12]

8. Ones / Six Sigma US. Requirements Traceability Matrix (RTM) in Agile and Compliance. *ones.com/blog/knowledge/requirement-traceability-matrix-agile-project-transparency; 6sigma.us/six-sigma-in-focus/requirements-traceability-matrix-rtm* [18]
9. Parasoft / Google Patents. Executable Traceability and Bidirectional Links to Source Code. *parasoft.com/blog/requirements-management-and-the-traceability-matrix; patents.google.com/patent/WO2017001417A1/en* [16]
10. MH4GF / Sverweij. Visualizing TypeScript dependencies using dependency-cruiser and Mermaid. *dev.to/mh4gf/visualize-typescript-dependencies-of-changed-files-in-a-pull-request-using-127j; github.com/MH4GF/dependency-cruiser-report-action* [24]
11. PSeitz / Remiscan. TypeScript dependency graph tools (ts-dependency-graph, dependency-cruiser VS Code Extension). *github.com/PSeitz/ts-dependency-graph; marketplace.visualstudio.com/items?itemName=remiscan.dependency-cruiser-ts* [22]
12. Project Manager / ClickUp. Deliverable Definition Sheet (DDS) template definition. *projectmanager.com/templates/project-deliverables-template; clickup.com/templates/project-deliverables-t-182201612* [8]
13. SitePoint / Goldbergyoni. BDD Gherkin Best Practices for Readability and Acceptance. *sitepoint.com/bdd-javascript-cucumber-gherkin; github.com/goldbergyoni/javascript-testing-best-practices* [10]
14. Council Fire / VGPB. Governance Checklists for Traceability and Contract Management. *councilfire.org/guides/traceability-metrics-checklist-for-value-chains* [26]
15. GitHub. About the Dependency Graph. *docs.github.com/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph* [20]

Works cited

1. Agile V-Model - digitalplaybook.org, accessed November 5, 2025, https://aiotplaybook.org/index.php?title=Agile_V-Model
2. About MADR | MADR - Architectural Decision Records, accessed November 5, 2025, https://adr.github.io/madr/
3. Examples | MADR - Architectural Decision Records, accessed November 5, 2025, https://adr.github.io/madr/examples.html
4. Work breakdown structure - Wikipedia, accessed November 5, 2025, https://en.wikipedia.org/wiki/Work_breakdown_structure
5. Work Breakdown Structure (WBS) - Basic Principles - PMI, accessed November 5, 2025, https://www.pmi.org/learning/library/work-breakdown-structure-basic-principles-4883
6. What Are Project Deliverables? | The Workstream - Atlassian, accessed November 5, 2025, https://www.atlassian.com/work-management/project-management/project-deliverables

7. What are project deliverables? Project management outcomes - Adobe for Business, accessed November 5, 2025, https://business.adobe.com/blog/basics/project-deliverables
8. Project Deliverables Template for Excel (Free Download), accessed November 5, 2025, https://www.projectmanager.com/templates/project-deliverables-template
9. Project Deliverables | Template by ClickUp™, accessed November 5, 2025, https://clickup.com/templates/project-deliverables-t-182201612
10. BDD in JavaScript: Getting Started with Cucumber and Gherkin - SitePoint, accessed November 5, 2025, https://www.sitepoint.com/bdd-javascript-cucumber-gherkin/
11. goldbergyoni/javascript-testing-best-practices: Comprehensive and exhaustive JavaScript & Node.js testing best practices (August 2025) - GitHub, accessed November 5, 2025, https://github.com/goldbergyoni/javascript-testing-best-practices
12. Test Automation of a Microservice using Cucumber, Java and ..., accessed November 5, 2025, https://medium.com/connectcd/test-automation-of-a-microservice-using-cucumber-java-and-openapi-c9bac9b7465d
13. Creating your first schema - JSON Schema, accessed November 5, 2025, https://json-schema.org/learn/getting-started-step-by-step
14. Exploring JSON Schema: Validation, Contract Testing, and Dynamic ..., accessed November 5, 2025, https://sohamnakhare.medium.com/exploring-json-schema-validation-contract-testing-and-dynamic-forms-c0472f4de2de
15. V Model vs Agile: What are the major differences? - KnowledgeHut, accessed November 5, 2025, https://www.knowledgehut.com/blog/agile/v-model-vs-agile
16. Requirements Management and the Traceability Matrix Templates & Examples - Parasoft, accessed November 5, 2025, https://www.parasoft.com/blog/requirements-management-and-the-traceability-matrix/
17. WO2017001417A1 - Tracing dependencies between development artifacts in a software development project - Google Patents, accessed November 5, 2025, https://patents.google.com/patent/WO2017001417A1/en
18. How a Requirement Traceability Matrix in Agile Boosts Project Transparency - ONES.com, accessed November 5, 2025, https://ones.com/blog/knowledge/requirement-traceability-matrix-agile-project-transparency/
19. Requirements Traceability Matrix: A Complete Guide for Project Success - Six Sigma, accessed November 5, 2025, https://www.6sigma.us/six-sigma-in-focus/requirements-traceability-matrix-rtm/
20. About the dependency graph - GitHub Docs, accessed November 5, 2025, https://docs.github.com/code-security/supply-chain-security/understanding-your-software-supply-chain/about-the-dependency-graph
21. How to Build a Dependency Graph: Streamlining Your Software Development Process, accessed November 5, 2025,

https://ones.com/blog/how-to-build-dependency-graph-streamline-software-development/
22. PSeitz/ts-dependency-graph: prints a dependency graph in dot/mermaid format for your typescript/react project - GitHub, accessed November 5, 2025, https://github.com/PSeitz/ts-dependency-graph
23. Dependency Cruiser TS - Visual Studio Marketplace, accessed November 5, 2025, https://marketplace.visualstudio.com/items?itemName=remiscan.dependency-cruiser-ts
24. Visualize TypeScript Dependencies of Changed Files in a Pull Request Using dependency-cruiser-report-action - DEV Community, accessed November 5, 2025, https://dev.to/mh4gf/visualize-typescript-dependencies-of-changed-files-in-a-pull-request-using-127j
25. MH4GF/dependency-cruiser-report-action: A GitHub Actions that report to visualize dependencies of changed files each pull requests., accessed November 5, 2025, https://github.com/MH4GF/dependency-cruiser-report-action
26. Contract governance checklist, accessed November 5, 2025, https://www.buyingfor.vic.gov.au/sites/default/files/2024-08/Contract-governance-checklist-goods-and-services.docx
27. Traceability Metrics Checklist for Value Chains - Council Fire - Guides, accessed November 5, 2025, https://www.councilfire.org/guides/traceability-metrics-checklist-for-value-chains/
28. BDD Anti-patterns - John Ferguson Smart, accessed November 5, 2025, https://johnfergusonsmart.com/slidedecks/bdd-anti-patterns/