ChatGPT

Deliverables-First Architecture Research & Blueprint

Executive Summary

Modern software initiatives often struggle with unclear requirements, ad-hoc task lists and undocumented design decisions. Deliverables-first architecture combats these issues by explicitly defining the products (deliverables) to be produced before deriving the work breakdown, recording architectural decisions as lightweight **Architecture Decision Records (ADRs)**, adopting **acceptance-first behaviour-driven development (ATDD/BDD)**, specifying **contract-first interfaces**, and maintaining end-to-end traceability through a **requirements traceability matrix (RTM)** and dependency graphs. This research synthesises industry guidance and provides a blueprint and exemplar demonstrating how to apply these techniques to a generic software development context. The literature review cites books, standards and articles that highlight benefits such as improved alignment, reduced rework and better quality by planning verification alongside development [1] [2]. The blueprint then shows how to plan a small feature—`Order creation`—using a product breakdown structure (PBS) leading to a work breakdown structure (WBS), ADRs capturing key decisions, an OpenAPI contract and Gherkin acceptance tests. A deliverable definition sheet (DDS) is created for each deliverable, linking acceptance criteria to tests, evidence and files. Traceability is maintained via an RTM and dependencies are visualised in a Mermaid graph. Finally, an adoption playbook offers step-by-step guidance for organisations to roll out deliverables-first practices.

Literature Review

Architecture Decision Records (ADRs)

An **architectural decision (AD)** is a justified design choice addressing a significant requirement, and an **ADR** documents that decision along with context, options, trade-offs and consequences [3]. Nygard popularised ADRs as short, sequentially numbered markdown files stored in version control; each ADR includes context, decision, and consequences, and may reference alternatives [4]. Once a decision is superseded, a new ADR should be added while keeping the old one for historical context [5]. Microsoft emphasises that ADRs should be written from project inception to provide a single source of truth and must include context, alternatives considered and justification [6] [7]. Properly maintained ADRs facilitate knowledge transfer and avoid repetition of past mistakes.

Product-Based Planning: PBS → WBS

Traditional work breakdown structures list tasks, which can lead to missing deliverables or extraneous tasks. Product-based planning proposes starting with a **Product Breakdown Structure (PBS)**—a hierarchical list of all products/deliverables (end products, intermediate deliverables, documentation) required to achieve the goal [8]. The PBS helps clarify scope, identify dependencies and quality criteria, and forms the basis for a **product flow diagram**. After the PBS, a **Work Breakdown Structure (WBS)** is derived, assigning work packages to create each product; this ensures every task is traceable back to a product and

prevents orphan tasks [2]. PRINCE2 materials stress that PBS creation should precede WBS creation, and both must align so that each product has work allocated and each work package produces a product [9]. PBS is particularly useful when teams must deliver complex systems or compliance documentation [10].

ATDD/BDD and Acceptance-First Development

Acceptance Test-Driven Development (ATDD) is a collaborative technique where business, development and testing stakeholders define acceptance criteria before coding begins [11]. The acceptance criteria are expressed in plain English and serve as high-level tests verifying that the feature meets user needs. **Behaviour-Driven Development (BDD)** focuses on system behaviour and uses the Gherkin language to describe scenarios with keywords such as *Feature*, *Scenario*, *Given*, *When* and *Then* [12]. AutomationPanda recommends writing one scenario per behaviour to keep tests comprehensible [13] and stresses that Gherkin should be understandable by non-technical stakeholders [14]. ATDD/BDD emphasises collaborative specification and early verification, aligning with the V-model's pairing of verification and validation stages [1]. The distinction between **Definition of Done** and **acceptance criteria** is that DoD is a checklist of quality requirements applicable to all stories, whereas acceptance criteria are specific tests for a particular story [15].

Contract-First Interfaces

An **API-first** or **contract-first** approach defines a machine-readable interface specification before implementation. Swagger promotes designing APIs as first-class products using OpenAPI definitions; this allows stakeholders to agree on endpoints and schemas before any code is written, enabling parallel development and reusable APIs [16]. Moesif notes that writing contracts upfront provides a blueprint for development, reduces miscommunication and supports automated code generation and testing [17] [18]. Postman further explains that API design-first (a flavour of API-first) requires collaborative design sessions resulting in human- and machine-readable contracts; this avoids the code-first pitfall where prototypes drive the API design [19]. Contract-first is particularly beneficial when multiple teams must integrate via defined interfaces or when regulatory compliance demands stable contracts.

V-Model and Planning Verification Early

The **V-model** pairs each development phase with a corresponding testing phase: business requirement analysis with acceptance testing, system design with system testing, and so on [20]. It emphasises that test planning should occur in parallel with development planning, leading to earlier defect detection and clearer acceptance criteria [21]. Though originally associated with waterfall projects, a "light V-model" can complement incremental development by aligning backlog items with test definitions and by ensuring that every deliverable has a verification strategy.

Requirements Traceability Matrix (RTM)

An **RTM** maps requirements to design, development and testing artifacts to ensure that all requirements are implemented and verified [22]. It supports **forward tracing** (from requirements to tests) and **backward tracing** (from tests or deliverables back to requirements), enabling compliance, change impact analysis and comprehensive test coverage [23]. The RTM typically includes columns for requirement IDs, descriptions, related test cases, design artifacts and implementation files [24]. A live RTM helps maintain alignment and can be automated using metadata in deliverable definitions and DoFD comments.

Dependency Graphs and Mapping

Dependency graphs visually represent how components, deliverables, tests and contracts depend on one another. Understanding dependencies helps teams manage change, identify critical paths and mitigate risks. PuppyGraph explains that dependency graphs model direct and transitive dependencies across compile-time, run-time and deployment layers [25]. Jit.io notes that mapping dependencies reveals vulnerabilities and assists security and compliance by making relationships explicit [26]. In a deliverables-first context, dependency graphs include edges from deliverables to tests, contracts and evidence, enabling clear traceability and risk analysis.

Comparison Matrix

Practice/ Technique	Purpose	Strengths	Trade-offs	Tooling	Adoption Tips	Good F
ADR (Architecture Decision Records)	Record significant design decisions with context, alternatives and consequences [4]	Provides historical rationale; avoids repeated debates; supports team autonomy; lightweight to maintain	Requires discipline to update; may become stale or ignored if not integrated into workflows	Markdown files in VCS; templates (MADR); tools like adr-tools or ADR-generator	Create ADR with each significant decision; number sequentially; link to affected deliverables/ contracts	Project evolvin archite multipl long-te mainte
PBS → WBS	Identify all products/ deliverables before breaking work into tasks [2]	Ensures scope clarity; prevents orphan tasks; enables quality criteria definition [8]	Initial overhead to identify products; may feel rigid for exploratory work	Simple diagrams/ spreadsheets; tools like MS Visio, Lucidchart; PRINCE2 product flow diagrams	Hold product-based planning workshop; involve stakeholders; derive WBS only after PBS approved	Project requiri compli fixed sc teams clarity deliver
ATDD/BDD	Define behaviour and acceptance criteria before coding [11]	Improves shared understanding; tests drive design; enhances collaboration; supports documentation [14]	Writing Gherkin scenarios requires skill; tests may be brittle if UI-coupled; may slow initial velocity	Tools like Cucumber.js, SpecFlow, Behave; Gherkin syntax [12]	Pair business analysts with developers when writing scenarios; keep scenarios atomic and business-oriented [13]	Regula domain accepta criteria explicit cross-fu teams

Practice/ Technique	Purpose	Strengths	Trade-offs	Tooling	Adoption Tips	Good F
Contract-First Interfaces	Define API or interface contract before implementing [16]	Enables parallel development; reduces integration bugs; allows auto-generation of code/docs; fosters reuse [18]	Upfront design effort; risk of over-design; contract changes require negotiation	OpenAPI/ Swagger for HTTP APIs; JSON Schema; Protocol Buffers; CLI specs (Cobra)	Host design reviews; treat contracts as versioned deliverables; test against contracts in CI	System multipl consun micros archite regulat public A
Light V-Model Planning	Pair every development activity with a planned verification activity [20]	Prevents late defect discovery; ensures test planning from the start; clarifies acceptance levels	May feel waterfall-like; less suited for rapid experimentation; requires discipline	Test strategies, RTM, acceptance test suites	During backlog refinement, define corresponding test(s) for each story; integrate test execution in CI	Project requiri quality compli safety- domain
RTM (Requirements Traceability Matrix)	Map requirements to deliverables, tests and evidence [22]	Ensures full coverage; supports change impact analysis; aids audits and compliance	Manual maintenance can be laborious; complex matrices can be hard to navigate	Spreadsheets; ALM tools (Jira, Azure DevOps); automated via YAML metadata and scripts	Use unique IDs; maintain links automatically via code annotations and CI; review regularly	Regula industr large p with m stakeh
Dependency Graphs	Visualise relationships and dependencies among components and deliverables [25]	Highlights critical paths; aids risk analysis; supports security and compliance [26]	May become cluttered for large systems; requires tooling to maintain	Graph visualisation (Mermaid, PlantUML); dependency analyzers	Generate automatically from build tools (npm ls, Maven) and metadata; embed in docs	Comple system many r micros compli needs

Blueprint

Assumptions

Because the user requested a generic context, the following default values are assumed:

Parameter	Value
Domain	General software development (no specific industry)
Example feature	*Order creation* — a service allows clients to create orders via an API
Target stack	TypeScript/Node.js 20
Test stack	Cucumber.js for ATDD, Jest for unit tests
Contract style	OpenAPI (3.0.3) with JSON Schema definitions
CI	GitHub Actions
Constraints	Moderate performance; secure coding practices; no special compliance constraints
Non-functional priorities	Operability and evolvability: clear structure, testability, maintainability

Product Breakdown Structure (PBS)

The PBS lists all products (deliverables) needed to implement the `Order creation` feature. Each deliverable includes a rationale.

ID	Deliverable	Rationale
DEL-001	API contract: `openapi.yaml`	Defines the HTTP interface for order creation and forms the contract for clients and servers; enables parallel development [16].
DEL-002	Acceptance test specification: `user_creates_order.feature`	Defines high-level behaviour using Gherkin; serves as ATDD spec and acceptance criteria [11].
DEL-003	Domain model & handler implementation: `src/api/handler.ts`	Implements the API endpoint and domain logic; must conform to the contract and acceptance tests.
DEL-004	Contract tests: `test/contract/openapi.test.ts`	Ensure the API implementation conforms to `openapi.yaml` by validating requests/responses.
DEL-005	Unit tests: `test/unit/handler.test.ts`	Validate domain logic independently of HTTP layer; support early detection of defects.
DEL-006	CI pipeline: `.github/workflows/ci.yml`	Automates installation, contract testing, acceptance testing and evidence collection.
DEL-007	Architectural decision log: `docs/adr/0001-contract-first.md`	Captures the decision to adopt contract-first and deliverables-first approaches; provides rationale and links.

ID	Deliverable	Rationale
DEL-008	Dependency graph documentation: `docs/deps.md`	Visualises dependencies among deliverables, tests and artifacts; aids risk analysis.
DEL-009	Requirements traceability matrix: `docs/rtm.csv`	Maps deliverables, acceptance criteria, tests, evidence and files for forward/backward traceability [22].

Work Breakdown Structure (WBS) Derived from PBS

All tasks derive directly from the PBS deliverables. No task exists without a corresponding product. Tasks are grouped by deliverable.

1. **Prepare API contract (DEL-001)** 1.1 Draft OpenAPI specification with endpoint and schema definitions. 1.2 Review contract with stakeholders; update if necessary. 1.3 Commit `openapi.yaml` to version control.

2. **Write acceptance specification (DEL-002)** 2.1 Identify scenarios for order creation. 2.2 Write Gherkin feature file `user_creates_order.feature` with scenario(s). 2.3 Review with stakeholders; refine language [14].

3. **Develop handler implementation (DEL-003)** 3.1 Scaffold Node project (package.json, tsconfig). 3.2 Implement domain logic for creating orders (`src/api/handler.ts`). 3.3 Add DoFD comment linking deliverable to tests and contract. 3.4 Write unit tests (`test/unit/handler.test.ts`) and ensure they fail initially.

4. **Develop contract tests (DEL-004)** 4.1 Write tests validating that requests and responses conform to `openapi.yaml` using a contract-test library (e.g., jest-openapi). 4.2 Ensure tests fail until implementation is complete.

5. **Develop acceptance tests (part of DEL-002)** 5.1 Write step definitions for Cucumber scenarios (e.g., `tests/acceptance/steps/order_steps.ts`). 5.2 Integrate with the API handler; implement test harness (supertest or similar). 5.3 Ensure acceptance tests initially fail (red).

6. **Set up CI pipeline (DEL-006)** 6.1 Configure GitHub Actions to install dependencies. 6.2 Add steps to run contract tests, unit tests and acceptance tests. 6.3 Publish evidence (test reports) to `reports/` directory.

7. **Document ADR (DEL-007)** 7.1 Write ADR capturing contract-first and deliverables-first decisions. 7.2 Include context, alternatives, decision and consequences [4].

8. **Create dependency graph (DEL-008)** 8.1 Use Mermaid to draw a graph linking deliverables to tests, contracts and files. 8.2 Document the graph in `docs/deps.md`.

9. **Prepare RTM (DEL-009)** 9.1 Assign unique IDs to deliverables (DEL-xxx), acceptance criteria (ACR-xxx), tests (TEST-xxx), evidence (EVID-xxx), files (FILE-xxx) and contracts (CT-xxx). 9.2 Populate `docs/rtm.csv` with rows mapping each deliverable to its tests/evidence/files.

Architectural Decision Records (ADRs)

docs/adr/0001-contract-first.md

```
# Contract-First & Deliverables-First Approach
*Status:* Accepted
*Date:* 2025-11-05

## Context

The project aims to deliver a generic order-creation API.  Multiple teams may
depend on the API, and we need clear contracts, testability and traceability.
Past projects suffered from ad-hoc task lists and undocumented decisions,
leading to rework and inconsistent interfaces.

## Decision

We will adopt a **contract-first** API design using OpenAPI 3.0.3, a
**deliverables-first** product breakdown, and an **acceptance-first**
development process.  ADRs will capture key decisions.  Every interface will
have a formal contract and associated contract tests.  Deliverables will be
defined via YAML sheets (DDS) linking acceptance criteria, evidence and files.
A traceability matrix will map requirements to tests and artifacts.

## Consequences

Positive:
• Enables parallel development and stakeholder alignment [16].
• Clear documentation of decisions improves maintainability and knowledge
transfer [4].
• RTM supports compliance and change impact analysis [22].
Negative:
• Upfront overhead to write contracts, ADRs and DDS.
• Requires discipline to keep ADRs and RTM updated.

## Alternatives Considered

• **Code-first API development:** easier initial prototyping but leads to
inconsistent interfaces and late integration issues [27].
• **Task-oriented planning:** simpler task lists but risks missing deliverables
and traceability [2].
```

```
## Related

Deliverables: DEL-001, DEL-002, DEL-003, DEL-004.
RTM IDs: see `docs/rtm.csv`.
Contracts: `contracts/openapi.yaml`.
```

Contracts (OpenAPI & JSON Schema)

contracts/openapi.yaml

```
openapi: 3.0.3
info:
  title: Order Service
  version: "0.1.0"
paths:
  /orders:
    post:
      summary: Create a new order
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateOrderRequest'
      responses:
        "201":
          description: Created
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Order'
components:
  schemas:
    CreateOrderRequest:
      type: object
      required:
        - customerId
        - items
      properties:
        customerId:
          type: string
          format: uuid
        items:
          type: array
          items:
            $ref: '#/components/schemas/OrderItem'
```

```
    Order:
      type: object
      required:
        - id
        - status
      properties:
        id:
          type: string
          format: uuid
        status:
          type: string
          enum: [CREATED, REJECTED]
    OrderItem:
      type: object
      required:
        - sku
        - qty
      properties:
        sku:
          type: string
        qty:
          type: integer
          minimum: 1
```

Acceptance Tests (Gherkin)

tests/acceptance/user_creates_order.feature

```
Feature: Order creation
  In order to track purchases
  As a customer
  I want to create an order

  Scenario: valid order is created
    Given a valid CreateOrderRequest payload
    When I POST it to /orders
    Then I receive 201 with an Order payload with status "CREATED"
```

tests/acceptance/steps/order_steps.ts (scaffold)

```
import { Given, When, Then } from '@cucumber/cucumber';
import request from 'supertest';
import { app } from '../../../src/app';

let response: request.Response;
```

```
let payload: any;

Given('a valid CreateOrderRequest payload', function () {
  payload = {
    customerId: '11111111-1111-1111-1111-111111111111',
    items: [ { sku: 'ABC123', qty: 1 } ]
  };
});

When('I POST it to /orders', async function () {
  response = await request(app).post('/orders').send(payload);
});

Then('I receive {int} with an Order payload with status {string}', function
(status: number, expectedStatus: string) {
  if (response.status !== status) throw new Error(`Expected ${status}, got $
{response.status}`);
  if (response.body.status !== expectedStatus) throw new Error(`Expected status
${expectedStatus}, got ${response.body.status}`);
});
```

Unit & Contract Test Scaffolds

test/contract/openapi.test.ts

```
import { matchers } from 'jest-openapi';
import request from 'supertest';
import { app } from '../../src/app';
import * as path from 'path';

expect.extend(matchers);

const apiSpecPath = path.join(__dirname, '../../contracts/openapi.yaml');

describe('OpenAPI contract', () => {
  beforeAll(() => {
    jestOpenAPI(apiSpecPath);
  });

  test('POST /orders matches contract', async () => {
    const res = await request(app)
      .post('/orders')
      .send({ customerId: '11111111-1111-1111-1111-111111111111', items: [{
sku: 'ABC123', qty: 1 }] });
    expect(res).toSatisfyApiSpec();
```

```
  });
});
```

test/unit/handler.test.ts

```
import { createOrder } from '../../src/api/handler';

describe('createOrder', () => {
  it('returns CREATED status for valid order', async () => {
    const result = await createOrder({ customerId: 'id', items: [{ sku: 'sku',
qty: 1 }] });
    expect(result.status).toBe('CREATED');
  });
});
```

Deliverable Definition Sheets (DDS)

Each deliverable has a YAML file under `deliverables/` describing its purpose, acceptance criteria, evidence and linked files.

deliverables/DEL-001.yaml

```
id: DEL-001
name: API contract
purpose: "Define the HTTP interface for order creation"
acceptance_criteria:
  - id: ACR-001
    given: A contract reviewer
    when: The OpenAPI file is opened
    then: All endpoints and schemas for order creation are defined and meet
stakeholder agreements
evidence:
  tests:
    - path: test/contract/openapi.test.ts
      proves: [ACR-001]
  reports: []
interfaces:
  consumes: []
  exposes:
    - name: createOrder
      contract: contracts/openapi.yaml
files_must_exist:
  - contracts/openapi.yaml
risks:
  - Schema changes could break clients
```

```
links:
  adr: docs/adr/0001-contract-first.md
  rtM_ids: [ACR-001, TEST-CT-001, CT-API]
```

deliverables/DEL-002.yaml

```
id: DEL-002
name: Acceptance test spec
purpose: "Define business behaviour for order creation in Gherkin"
acceptance_criteria:
  - id: ACR-002
    given: A valid CreateOrderRequest
    when: It is posted to /orders
    then: The system responds with 201 and status CREATED
evidence:
  tests:
    - path: tests/acceptance/user_creates_order.feature
      proves: [ACR-002]
  reports: []
interfaces:
  consumes:
    - name: createOrder
      contract: contracts/openapi.yaml
  exposes: []
files_must_exist:
  - tests/acceptance/user_creates_order.feature
  - tests/acceptance/steps/order_steps.ts
risks:
  - Ambiguity in Gherkin may mislead developers
links:
  adr: docs/adr/0001-contract-first.md
  rtM_ids: [ACR-002, TEST-ACPT-001]
```

deliverables/DEL-003.yaml

```
id: DEL-003
name: Handler implementation
purpose: "Implement the createOrder API and domain logic"
acceptance_criteria:
  - id: ACR-003
    given: The handler receives a valid request
    when: It processes the order
    then: It returns an Order object conforming to the schema with status
CREATED
evidence:
```

```
  tests:
    - path: test/unit/handler.test.ts
      proves: [ACR-003]
    - path: test/contract/openapi.test.ts
      proves: [ACR-003]
    - path: tests/acceptance/user_creates_order.feature
      proves: [ACR-003]
  reports: []
interfaces:
  consumes:
    - name: createOrder
      contract: contracts/openapi.yaml
  exposes: []
files_must_exist:
  - src/api/handler.ts
risks:
  - Incorrect domain logic may cause status mismatches
links:
  adr: docs/adr/0001-contract-first.md
  rtM_ids: [ACR-003, TEST-UNIT-001, TEST-CT-001, TEST-ACPT-001]
```

Requirements Traceability Matrix (RTM)

The RTM maps deliverables, acceptance criteria, tests, evidence and files. Each row links an artifact to its related items. The CSV can be generated automatically from DDS metadata.

docs/rtm.csv

```
type,id,links_to,artifact
Deliverable,DEL-001,ACR-001;TEST-CT-001;FILE-contracts/
openapi.yaml,deliverables/DEL-001.yaml
Deliverable,DEL-002,ACR-002;TEST-ACPT-001;FILE-tests/acceptance/
user_creates_order.feature,deliverables/DEL-002.yaml
Deliverable,DEL-003,ACR-003;TEST-UNIT-001;TEST-CT-001;TEST-ACPT-001;FILE-src/
api/handler.ts,deliverables/DEL-003.yaml
Acceptance,ACR-001,TEST-CT-001,deliverables/DEL-001.yaml
Acceptance,ACR-002,TEST-ACPT-001,deliverables/DEL-002.yaml
Acceptance,ACR-003,TEST-UNIT-001;TEST-CT-001;TEST-ACPT-001,deliverables/
DEL-003.yaml
Test,TEST-CT-001,EVID-CT-001,test/contract/openapi.test.ts
Test,TEST-UNIT-001,EVID-UNIT-001,test/unit/handler.test.ts
Test,TEST-ACPT-001,EVID-ACPT-001,tests/acceptance/user_creates_order.feature
Evidence,EVID-CT-001,,reports/contract/latest.json
Evidence,EVID-UNIT-001,,reports/unit/latest.json
Evidence,EVID-ACPT-001,,reports/acpt/latest.json
File,FILE-contracts/openapi.yaml,DEL-001,contracts/openapi.yaml
```

```
File,FILE-tests/acceptance/user_creates_order.feature,DEL-002,tests/acceptance/
user_creates_order.feature
File,FILE-src/api/handler.ts,DEL-003,src/api/handler.ts
Contract,CT-API,TEST-CT-001,contracts/openapi.yaml
```

Dependency Graph

docs/deps.md

```
graph TD
  DEL-001[API Contract (openapi.yaml)] --> TEST-CT-001[Contract Tests]
  DEL-002[Acceptance Spec] --> TEST-ACPT-001[Acceptance Tests]
  DEL-003[Handler Implementation] --> TEST-UNIT-001[Unit Tests]
  DEL-003 --> TEST-CT-001
  DEL-003 --> TEST-ACPT-001
  TEST-CT-001 --> EVID-CT-001[Contract Test Evidence]
  TEST-UNIT-001 --> EVID-UNIT-001[Unit Test Evidence]
  TEST-ACPT-001 --> EVID-ACPT-001[Acceptance Test Evidence]
  FILE-src_api_handler[handler.ts] --> DEL-003
  FILE-contracts_openapi[openapi.yaml] --> DEL-001
  FILE-user_creates_order[user_creates_order.feature] --> DEL-002
```

File-Level Definition of File Deliverable (DoFD) Comments

Add the following comment at the top of each implementation file to declare its deliverable, acceptance, tests and contracts:

src/api/handler.ts

```
/* DoFD:
deliverable: DEL-003
satisfies_acceptance: [ACR-003]
tests_must_pass:
  - test/unit/handler.test.ts
  - test/contract/openapi.test.ts
  - tests/acceptance/user_creates_order.feature
contracts_must_pass:
  - contracts/openapi.yaml
*/

export interface CreateOrderRequest {
  customerId: string;
  items: Array<{ sku: string; qty: number }>;
}
```

```typescript
export interface Order {
  id: string;
  status: 'CREATED' | 'REJECTED';
}

// Implementation stub
export async function createOrder(req: CreateOrderRequest): Promise<Order> {
  return { id: 'generated-id', status: 'CREATED' };
}

// Express handler or Fastify handler would wrap this function in the real
implementation
```

Repository Structure

```
.
├── contracts/
│   └── openapi.yaml
├── deliverables/
│   ├── DEL-001.yaml
│   ├── DEL-002.yaml
│   └── DEL-003.yaml
├── docs/
│   ├── adr/
│   │   └── 0001-contract-first.md
│   ├── deps.md
│   └── rtm.csv
├── src/
│   └── api/
│       └── handler.ts
├── tests/
│   ├── acceptance/
│   │   ├── user_creates_order.feature
│   │   └── steps/
│   │       └── order_steps.ts
│   ├── contract/
│   │   └── openapi.test.ts
│   └── unit/
│       └── handler.test.ts
└── .github/
    └── workflows/
        └── ci.yml
```

CI Plan (.github/workflows/ci.yml)

```
name: CI
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: 20
      - run: npm ci
      - name: Unit tests
        run: npm run test:unit
      - name: Contract tests
        run: npm run test:contract
      - name: Acceptance tests
        run: npm run test:acceptance
      - name: Publish evidence
        run: |
          mkdir -p reports/contract reports/unit reports/acpt
          npm run test:unit -- --json > reports/unit/latest.json || true
          npm run test:contract -- --json > reports/contract/latest.json || true
          npm run test:acceptance -- --format json:reports/acpt/latest.json ||
true
```

How to Run Locally

1. **Install dependencies:** run `npm install` in the project root.
2. **Generate TypeScript build (if necessary):** run `npm run build` (not required in this stub because code runs directly in Node via ts-node/register in tests).
3. **Run unit tests:** `npm run test:unit`.
4. **Run contract tests:** `npm run test:contract` (requires `jest-openapi`).
5. **Run acceptance tests:** `npm run test:acceptance` (runs Cucumber.js).
6. **View dependency graph:** open `docs/deps.md` to see the Mermaid diagram; use a Mermaid viewer to render it.
7. **Check RTM:** open `docs/rtm.csv` and verify that each deliverable has tests and evidence.

Adoption Playbook

1. **Educate the team on deliverables-first principles.** Present the rationale and benefits from the literature review—recording decisions via ADRs, defining products before tasks, and using contract- and acceptance-first approaches.

2. **Start with a pilot project.** Choose a manageable feature (like order creation) and run a product-based planning workshop. Identify deliverables and create a PBS. Use the provided templates (DDS, ADR, RTM) to document them.
3. **Set up repository scaffolding.** Create folders for `contracts`, `deliverables`, `docs/adr`, `src`, `tests` and `.github/workflows`. Include README instructions.
4. **Adopt contract-first API design.** Use OpenAPI or another contract language to define interfaces. Review contracts with stakeholders. Add contract tests in CI.
5. **Write acceptance criteria before coding.** For each deliverable, define acceptance criteria in Gherkin; review with stakeholders; check for unambiguous language [14].
6. **Implement using TDD/BDD.** Write failing unit tests and acceptance tests; implement code to make tests pass. Keep scenarios atomic and business-focused [13].
7. **Maintain ADRs and DDS.** For every significant decision, write an ADR with context, alternatives and consequences. Update DDS files whenever deliverables, acceptance criteria or tests change.
8. **Automate traceability.** Generate the RTM automatically from DDS metadata using scripts. Verify that each deliverable links to tests and evidence; update DoFD comments in code files.
9. **Visualise dependencies.** Generate and maintain dependency graphs as part of documentation. Use tools to update graphs when new modules or contracts appear.
10. **Iterate and scale.** After the pilot, refine the process based on feedback. Gradually apply deliverables-first architecture across projects. Provide training, templates and automation to reduce overhead.

References

1. Microsoft. *Documenting architecture decisions.* (2023) [6] [7].
2. ADR Guide. *What is an Architecture Decision Record?* (2020) [3].
3. M. Nygard. *Documenting Architecture Decisions.* (2011) [4] [5].
4. Twproject. *PBS vs WBS: what is the difference?* (2022) [28] [2].
5. PRINCE2 Wiki. *Product-based Planning.* (n.d.) [8] [9].
6. The Knowledge Academy. *Product Breakdown Structure (PBS).* (2021) [10].
7. GeeksforGeeks. *Involvement of testers in TDD, BDD & ATDD.* (2022) [11] [29] [30].
8. AutomationPanda. *Write good Gherkin.* (2017) [14] [13].
9. Cucumber. *Gherkin Reference.* (2020) [12] [31].
10. Visual Paradigm. *Definition of Done vs Acceptance Criteria.* (2021) [15] [32] [33].
11. Swagger. *API First.* (2023) [16] [34].
12. Moesif. *Contract-first API development.* (2023) [17] [18] [35].
13. Postman. *API-first vs code-first.* (2024) [36] [27].
14. BuiltIn. *V-Model.* (2022) [21] [20] [37].
15. Webomates. *Requirements Traceability Matrix.* (2021) [22] [23].
16. Abstracta. *Requirements Traceability Matrix.* (2020) [24].
17. PuppyGraph. *Software Dependency Graphs.* (2023) [25] [38].
18. Jit.io. *Application Dependency Mapping.* (2022) [26].

1 20 21 37 What Is the V-Model? (Definition, Examples) | Built In
https://builtin.com/software-engineering-perspectives/v-model

2 28 Work breakdown structure vs Product breakdown structure
https://twproject.com/blog/work-breakdown-structure-vs-product-breakdown-structure/

3 Architectural Decision Records (ADRs) | Architectural Decision Records
https://adr.github.io/

4 5 Documenting Architecture Decisions
https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions.html

6 7 Architecture decision record - Microsoft Azure Well-Architected Framework | Microsoft Learn
https://learn.microsoft.com/en-us/azure/well-architected/architect-role/architecture-decision-record

8 9 Plans :: PRINCE2® wiki
https://prince2.wiki/practices/plans/

10 Product Breakdown Structure: Benefits, Examples & Templates
https://www.theknowledgeacademy.com/blog/product-breakdown-structure/

11 29 30 How The Testers Are Involved In TDD, BDD & ATDD Techniques? - GeeksforGeeks
https://www.geeksforgeeks.org/software-testing/how-the-testers-are-involved-in-tdd-bdd-atdd-techniques/

12 31 Cucumber
https://cucumber.io/docs/gherkin/reference/

13 14 BDD 101: Writing Good Gherkin | Automation Panda
https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/

15 32 33 Definition of Done vs Acceptance Criteria
https://www.visual-paradigm.com/scrum/definition-of-done-vs-acceptance-criteria/

16 34 Understanding the API-First Approach to Building Products
https://swagger.io/resources/articles/adopting-an-api-first-approach/

17 18 35 Mastering Contract-First API Development: Key Strategies and Benefits | Moesif Blog
https://www.moesif.com/blog/technical/api-development/Mastering-Contract-First-API-Development-Key-Strategies-and-Benefits/

19 27 36 What is API-first? The API-first Approach Explained | Postman
https://www.postman.com/api-first/

22 23 Requirements traceability matrix in an Agile process – Webomates
https://www.webomates.com/blog/software-requirement-metrics/9-reasons-why-requirements-traceability-is-important-in-agile/

24 Requirements Traceability Matrix: Your QA Strategy | Abstracta
https://abstracta.us/blog/testing-strategy/requirements-traceability-matrix-your-qa-strategy/

25 38 Software Dependency Graphs: Definition, Use Cases, and Implementation
https://www.puppygraph.com/blog/software-dependency-graph

26 A Developer’s Guide to Dependency Mapping | Jit
https://www.jit.io/resources/app-security/a-developers-guide-to-dependency-mapping