Promoting **code**, **pipelines**, and **data models** between environments (Dev → Test → Prod) is done using **Foundry's "Workspaces", "Branches", "Code Repositories"**, and optionally **Lifecycle Management (LCM)** features, depending on how strict your setup is.

Let's walk through **how promotion works in practice**, including the **mechanics, tools, and best practices**.

🧱 Environments in Foundry

Foundry environments typically follow a structure like:

- **Development (Dev):** For building, iterating, and debugging
- **Test (QA/UAT):** For integration testing, staging
- **Production (Prod):** Stable, certified, auditable

Each environment can be modeled as a **workspace**, or **project variant** (e.g., branches or tiles), depending on governance level.

🔁 Promotion Strategy Overview

Element	**Tool/Mechanism**	**Notes**
Code (SQL/Python/Transformations)	**Code Repository & Branching**	Similar to Git workflows
Data Models / Ontology	**Object Model versions & sync tools**	Models can be versioned, validated
Datasets	**Templated output paths or Environment Variables**	Output can be routed by environment
Dashboards / Apps	**Slate branches** or **App configuration per env**	UI separated per environment
Pipeline Logic	**LCM Tooling** (for governed orgs)	For managing promotion across multiple environments with traceability

---

🔧 1. Code Promotion with Code Repositories

🛠️ Process:

1. Develop code in a **Dev branch** (e.g., feature/sales_metrics_v2)
2. Test in Dev Workspace
3. Merge into main or release branch once validated
4. Tag a **release** if needed
5. Sync that branch into **Test or Prod Workspaces**

This can be done through:

- **Foundry Git Repos** (with built-in version control)
- Or using **external GitHub + Mirror Sync** integrations

Think of it just like Git: write code → test → merge → promote.

📦 2. Data Model / Ontology Promotion

For Foundry's **Object Models**:

- Ontologies (Object types, Actions, Datasets) are **version-controlled**
- You can create **draft versions**, validate them, then publish to Prod

Process:

- Edit model in Dev
- Validate object references
- Push to version (e.g., v2.1)
- Deploy the new model into the **Production Workspace**

Tools: Object Explorer, Model Sync, and API-based deployment

🗃️ 3. Dataset Promotion & Isolation

Each environment should use its **own datasets**:

- Output paths: use environment-based paths or dataset prefixes (dev/sales_summary, prod/sales_summary)
- Use **parameters** or **environment variables** in pipelines to route data correctly

Example:

env = context.params.get("env", "dev")

output_path = f"/{env}/sales_summary"

This ensures Dev data never mixes with Prod.

📅 4. Orchestration & Scheduling Per Environment

Schedules should be **tied to environment**:

- Dev: manual or ad-hoc runs
- Test: runs with test inputs or mock parameters
- Prod: scheduled daily/hourly with alerts

Foundry lets you **copy a pipeline** from one workspace to another, retaining logic but allowing you to retarget inputs, outputs, and schedule.

🔐 5. Lifecycle Management (LCM) for Promotion

Optional — for advanced governance setups

Foundry has a **Lifecycle Management (LCM)** tool:

- Define **Promotion Pipelines**
- Track approvals, versions, data dependencies
- Push transformations + ontology + applications across environments in a **controlled way**

✅ Ideal for regulated industries (finance, healthcare, defense)

🔁 Example: Dev → Test → Prod Flow

1. Code repo: create feature in Dev → test with dev datasets
2. Merge to release branch → sync to Test workspace
3. Validate pipeline using test datasets
4. Promote to Prod workspace
5. Update Prod ontology + dashboard config
6. Enable production schedule

All this can be documented and tracked via Foundry's **Promotion Tracking**, so you know:

- Who promoted what
- Which version is running where
- What datasets, models, and dashboards were affected

✅ Best Practices

Tip	Why
Use consistent naming across environments	Avoid confusion (e.g., sales_summary_dev, sales_summary_prod)
Parameterize pipelines and configs	Avoid hardcoding paths, thresholds
Tag and document dataset versions	Helps trace issues if rollback needed
Use approval workflows if supported	Especially in regulated industries
Regularly clean up dev/test outputs	Avoid clutter and cost

Here's a **template structure** you can adapt directly in Palantir Foundry, organized by environment and designed to be plug-and-play. I'll provide both a **logical layout** and a **sample JSON-style spec** that mirrors a workbook structure you might implement.

🧱 Structure: Project Layout by Environment

Folder Layout:

```
/project-root/
|
 ├── /dev/
|    ├── transformations/
|    ├── datasets/
|    └── dashboards/
|
 ├── /test/
```

```
|   ├── transformations/
|   ├── datasets/
|   └── dashboards/
|
├── /prod/
|   ├── transformations/
|   ├── datasets/
|   └── dashboards/
|
└── /shared/
    ├── reference_data/
    ├── config/
    └── code_templates/
```

Use shared/ for stable, reusable artifacts (e.g., reference tables, global configs, common code blocks).

🧰 Template: Parameterized Transformation Node (Python / PySpark)

```python
# Load params from context
params = context.params
env = params.get("env", "dev")  # dev, test, prod

# Use environment-specific logic
input_path = f"/{env}/datasets/raw_sales_data"
output_path = f"/{env}/datasets/sales_summary"

df = spark.read.parquet(input_path)
# ... do transformations ...
df.write.mode("overwrite").parquet(output_path)
```

🧰 Template: SQL Transformation Node

```
-- Use templated parameters passed in the transformation config
SELECT
  region,
  category,
  DATE(transaction_ts) AS transaction_date,
  COUNT(*) AS transactions,
  SUM(transaction_amount) AS total_sales
FROM `${input_sales_data}`
WHERE transaction_date >= '${start_date}'
GROUP BY region, category, DATE(transaction_ts)
```

Parameters defined in UI:

```
{
  "input_sales_data": "/dev/datasets/raw_sales_data",
  "start_date": "${TODAY.minusDays(7)}"
}
```

🔄 Promotion Template: Foundry Manifest (LCM Spec Style)

For orgs using Lifecycle Management (LCM), here’s a simplified JSON manifest format to promote components between environments:

```
{
  "promotionSpec": {
    "sourceEnvironment": "dev",
    "targetEnvironment": "test",
    "components": [
      {
        "type": "Transformation",
        "name": "sales_summary_transform"
      },
      {
        "type": "Dataset",
        "name": "sales_summary"
      },
      {
        "type": "OntologyModel",
        "name": "SalesAnalyticsOntology",
        "version": "v2.0"
      },
      {
        "type": "SlateApp",
        "name": "SalesDashboard"
      }
    ]
  }
}
```

This format mirrors what a controlled promotion system might read to perform deployments.

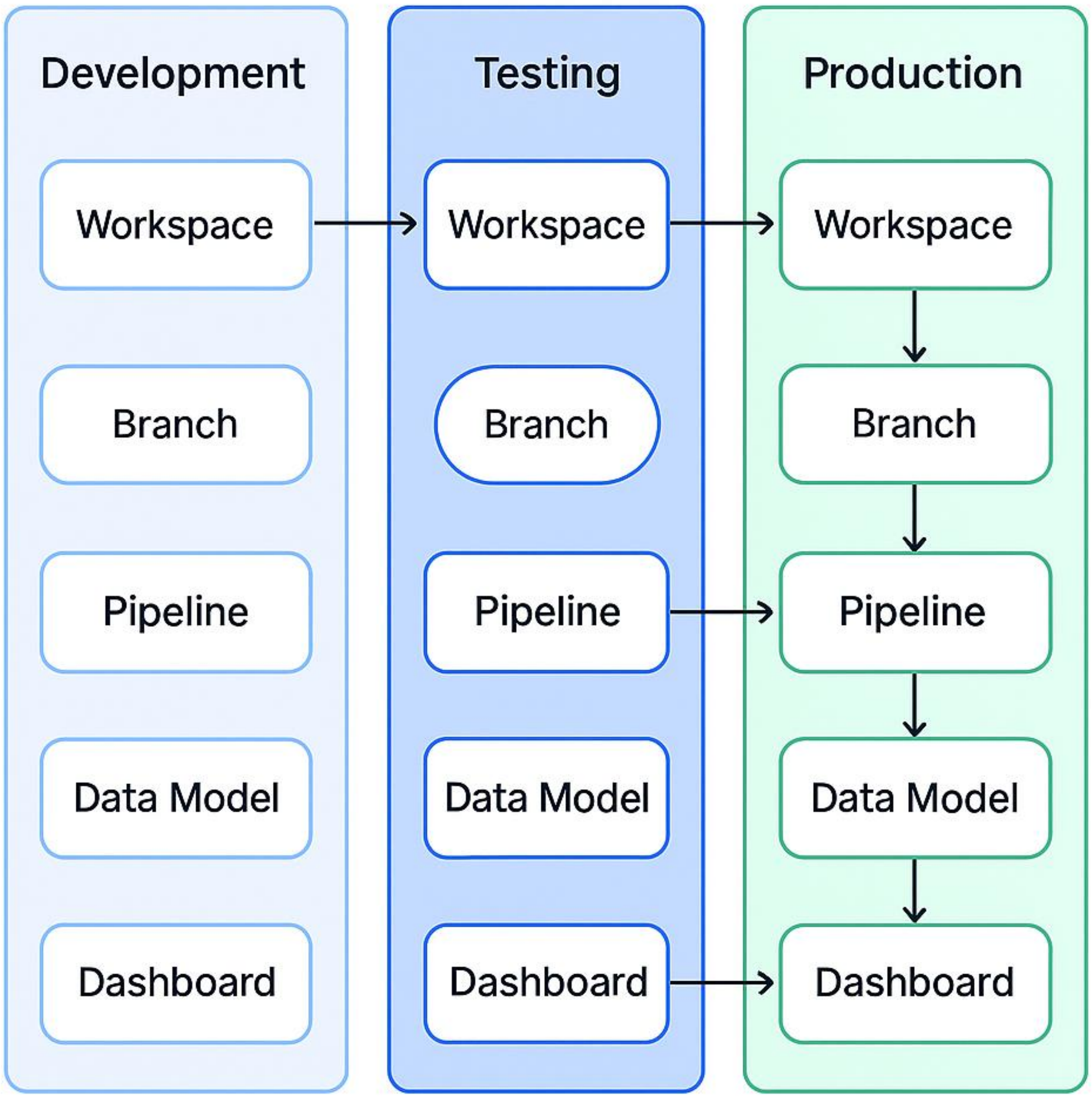
✅ Workbook Template (Transform Graph)

Node	Type	Description
raw_sales_data	Input Dataset	Pulled from /shared/ref/sales_data_raw
cleaned_sales_data	SQL	Cleans & formats raw
sales_enriched	Python	Joins with product and customer ref
sales_summary	SQL	Aggregates into KPIs
sales_summary_output	Output Dataset	Written to /${env}/datasets/sales_summary

You can save this as a **Graph Template** in Foundry and reuse across environments.

You can find Foundry-compatible pipeline template:foundry_sales_pipeline_template.json

In your github repository. You can import or adapt this for use in Foundry projects, workbooks, or promotion pipelines.


Development
Testing
Production
Workspace
Workspace
Workspace
Branch
Branch
Branch
Pipeline
Pipeline
Pipeline
Data Model
Data Model
Data Model
Dashboard
Dashboard
Dashboard