Building the Next Youtube

IN YOUR DORM

What is the **goal** of this **session**?

And now a word

FROM OUR SPONSOR

https://cloud.google.com/free/docs/aws-azure-gcp-service-comparison

Prelude Product

As software engineers, **why do we exist?**

We **deliver** high-quality software...

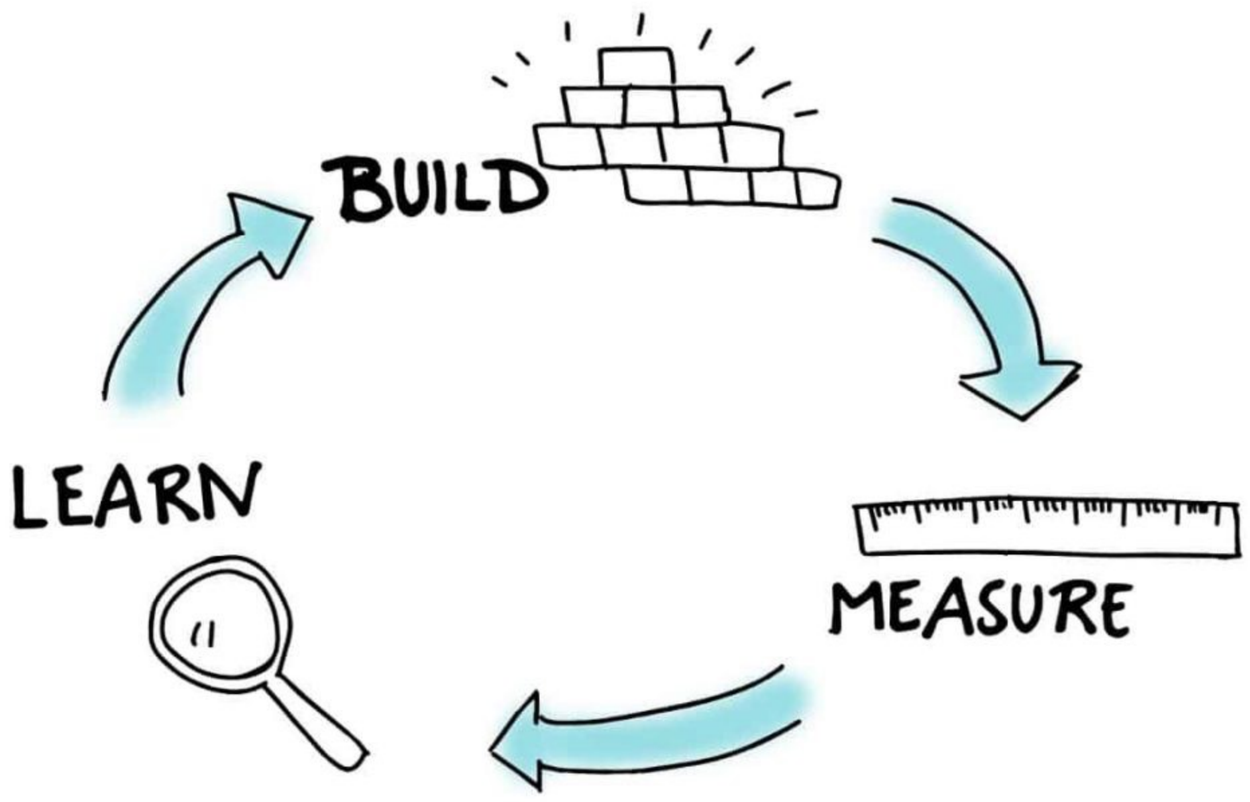
That solves a
real problem
for our customers

How do we **know**
anything?

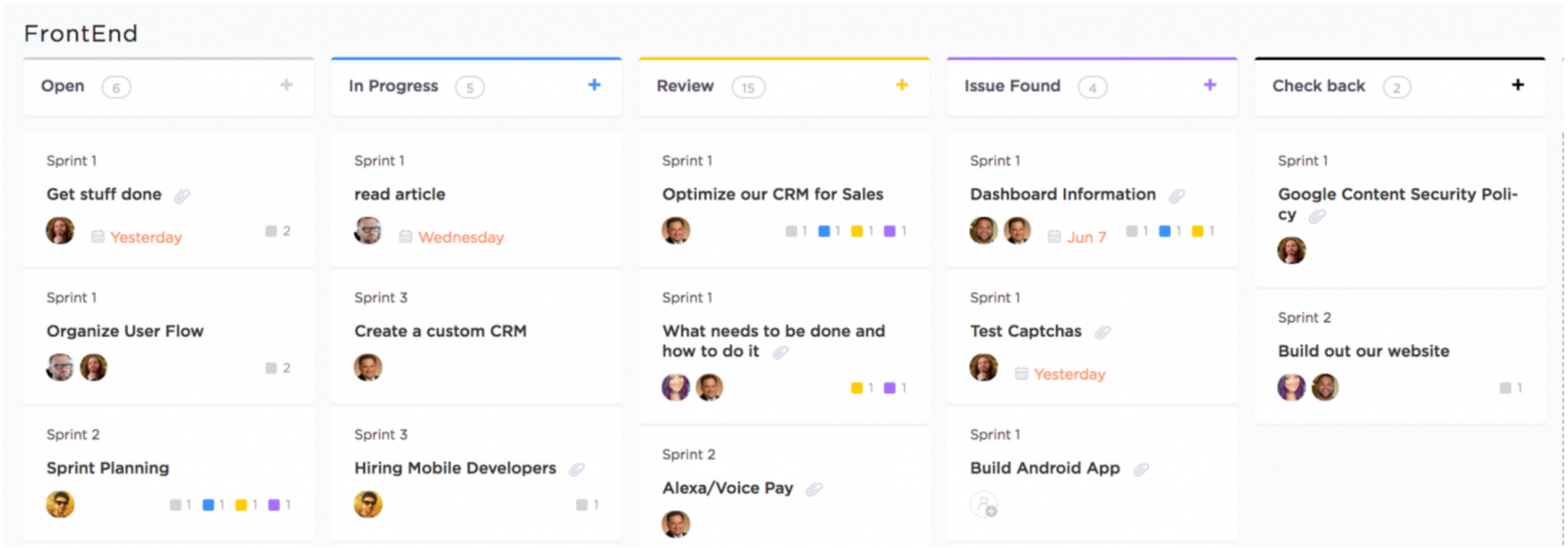
How do we know that we are building **the right thing**?

Feedback Loops

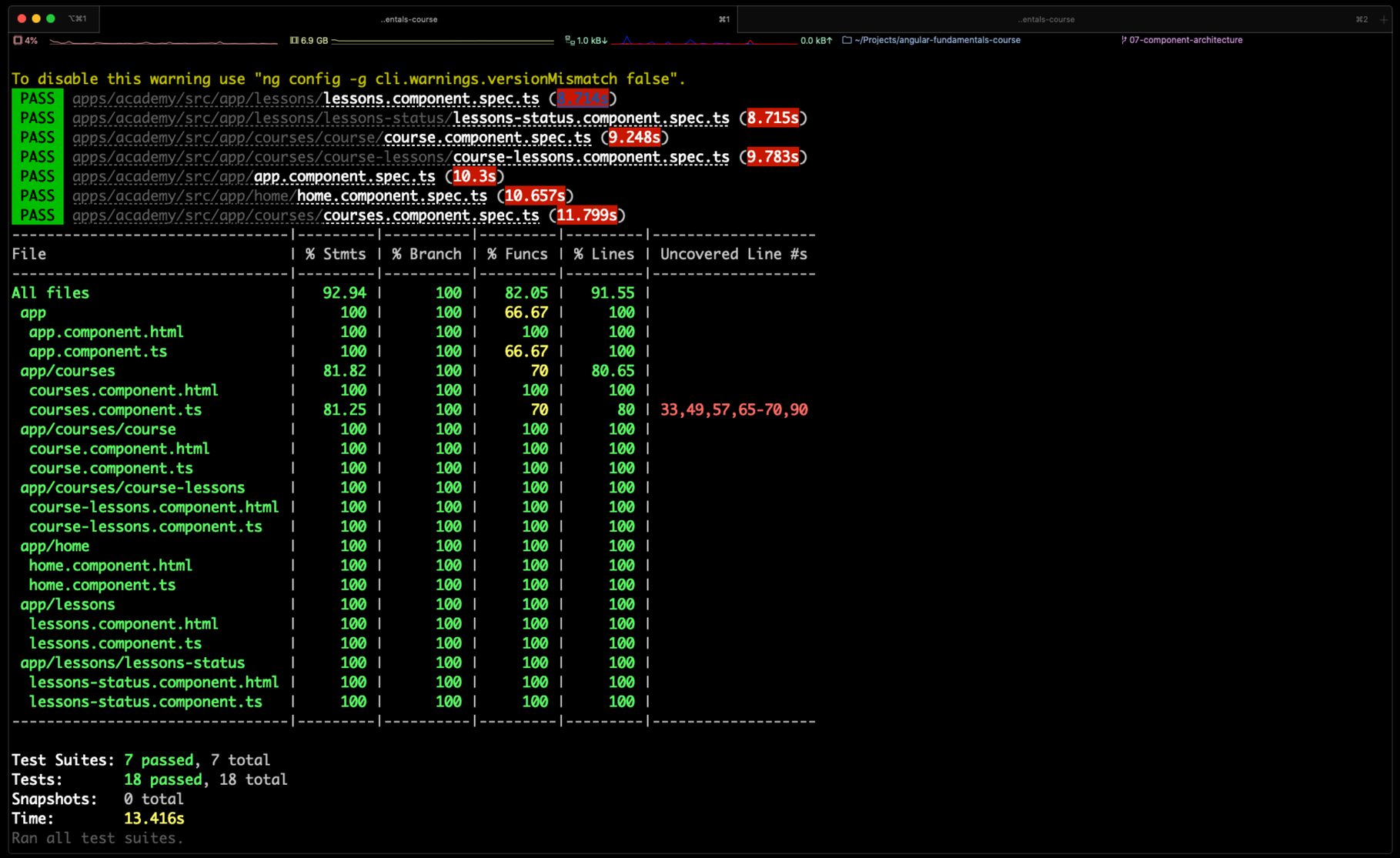
Classic Feedback Loop



Classic Agile Board



Classic Love Story



How do feedback loops accelerate delivery?

Make it work.
Make it right.
Make it fast.

- Kent Block

Make it work.

Make it known.

Make it right.

Make it fast.

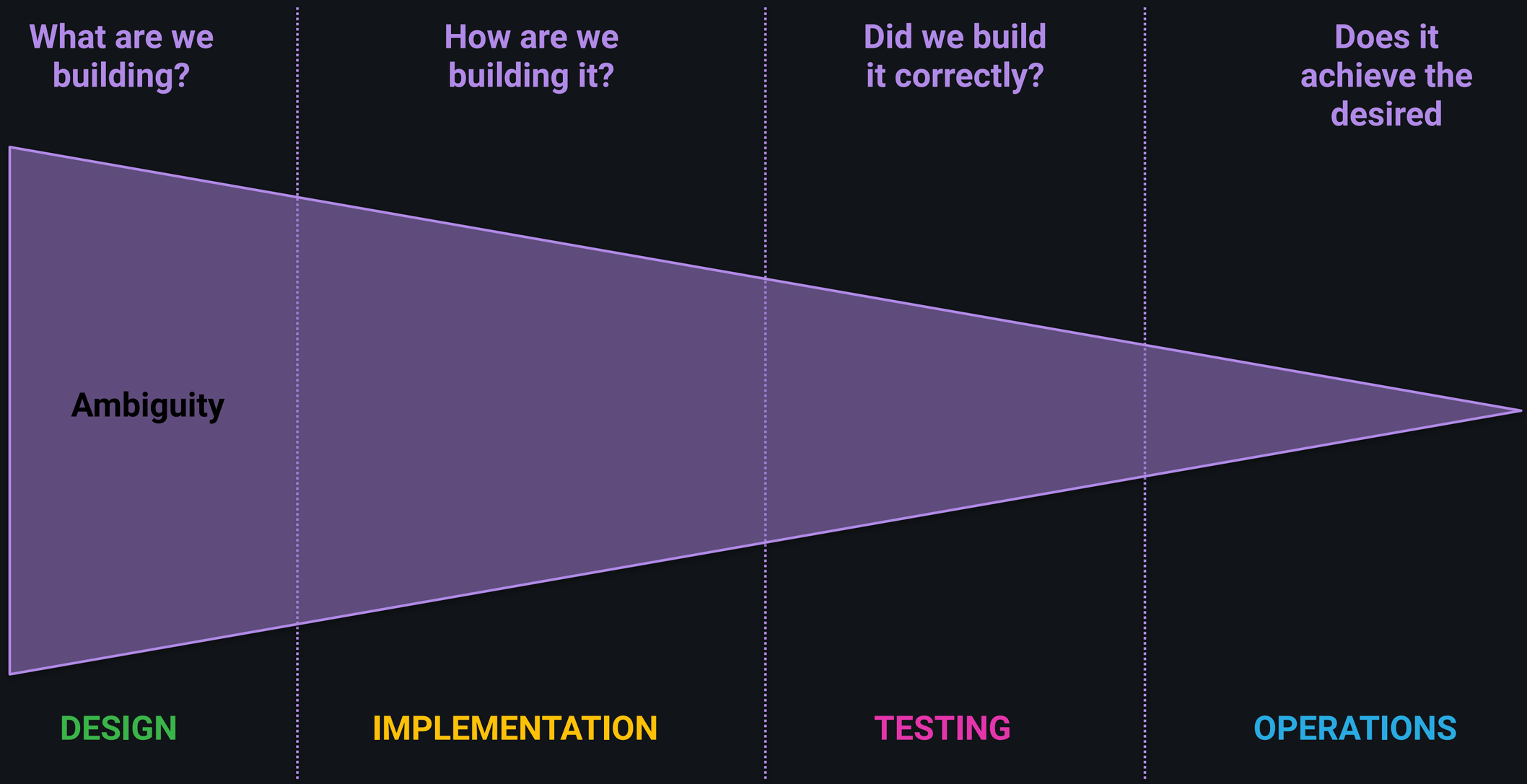
- Lukas Ruebbelke

How fast can we put what we are building **in the customer's hands**?

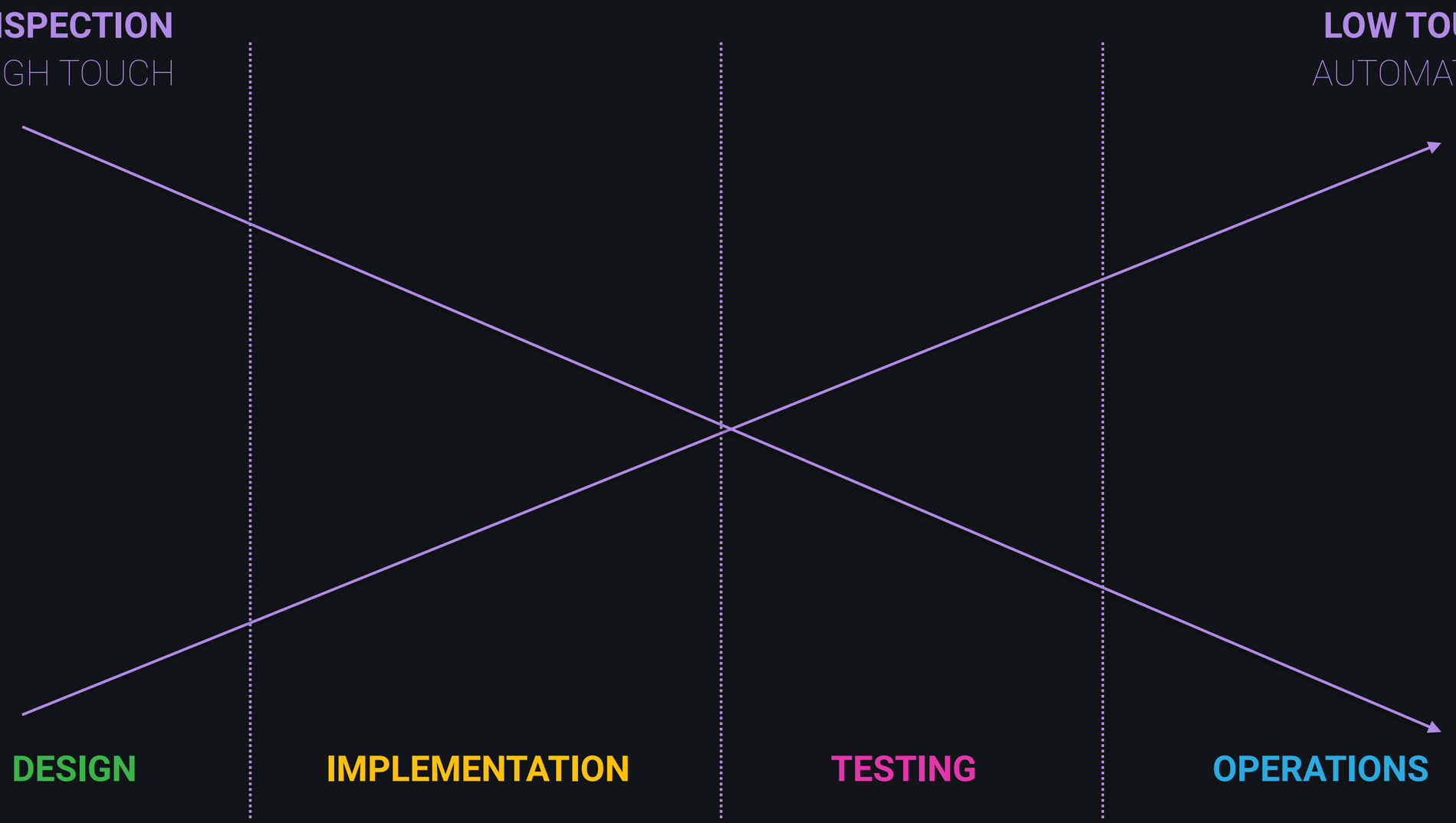
Home Team Away Team

Customers **will tell us** if we are building the **right thing**

The Software Development Lifecycle



The Testing Spectrum



Abstractions

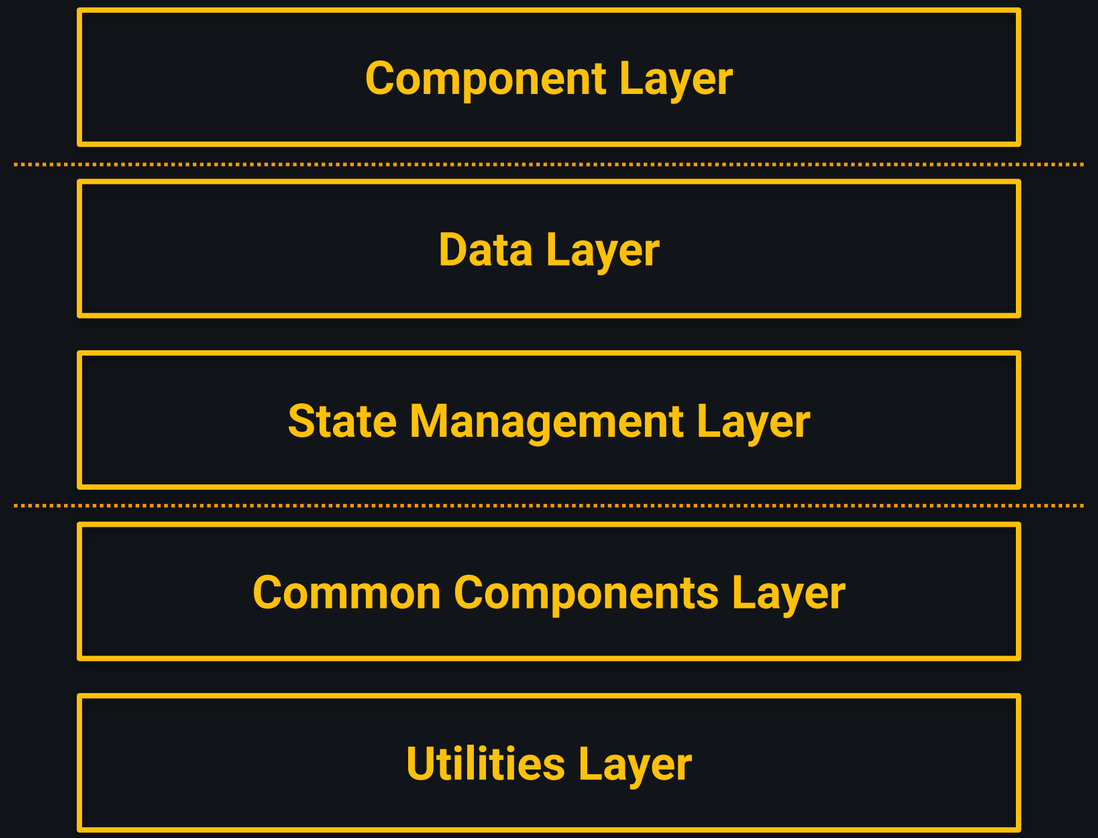
Abstraction is a technique for **arranging complexity** of computer systems

Abstraction works by **identifying** levels of complexity...

And **suppressing those details** to the **lowest possible level** in which they can still be effective

Abstraction Layers

Abstraction Layers



Coupling is the result of one element in code depending on another

Cohesion is the result of one element of code working with another element of code to serve a common purpose

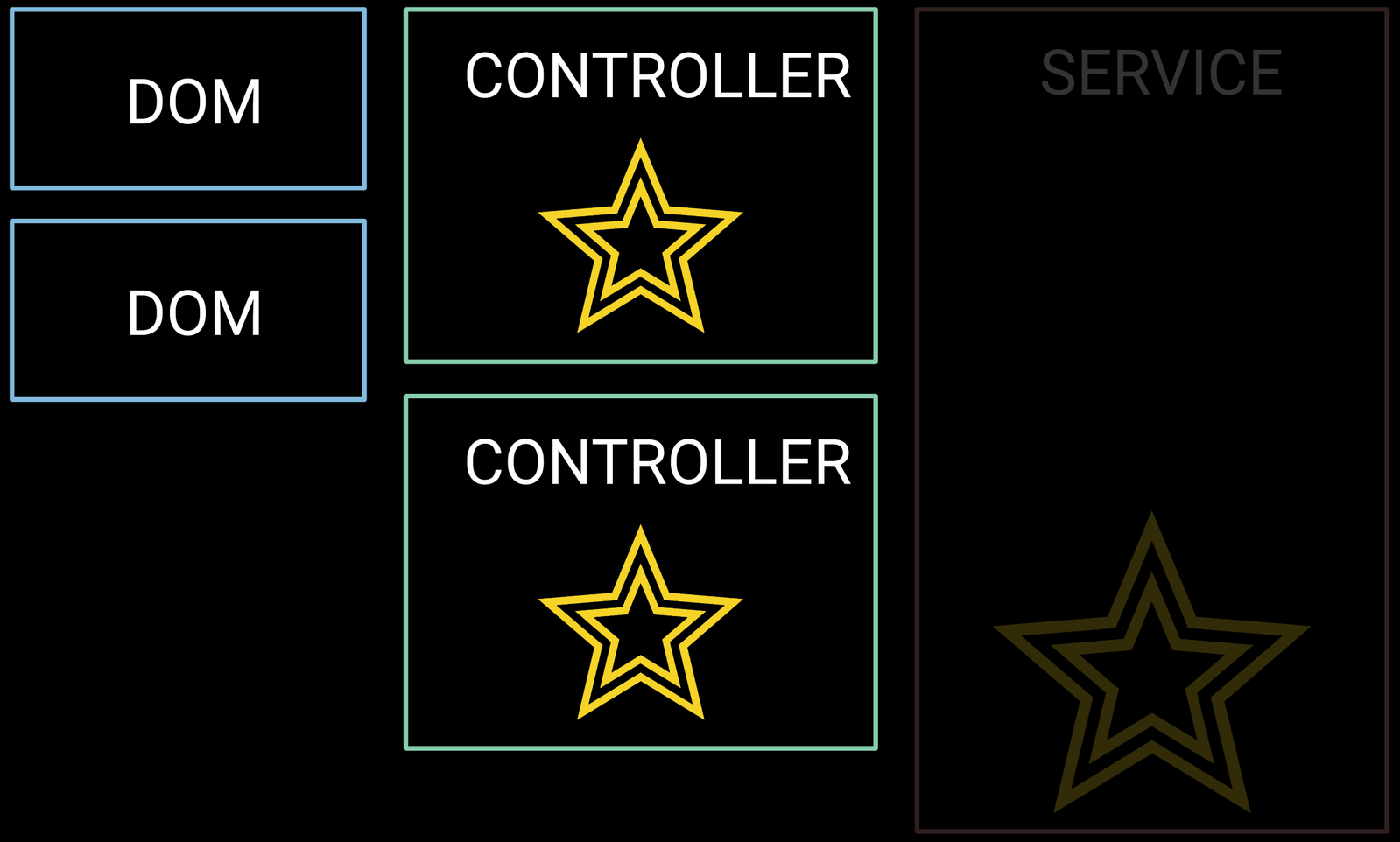

```
$( "form" ).submit(function( event ) {
  if ( $( "input:first" ).val() === "correct" ) {
    $( "span" ).text( "Validated..." ).show();
    return;
  }

  $( "span" ).text( "Not
valid!" ).show().fadeOut( 1000 );
  event.preventDefault();
});
```

jQuery “Application”

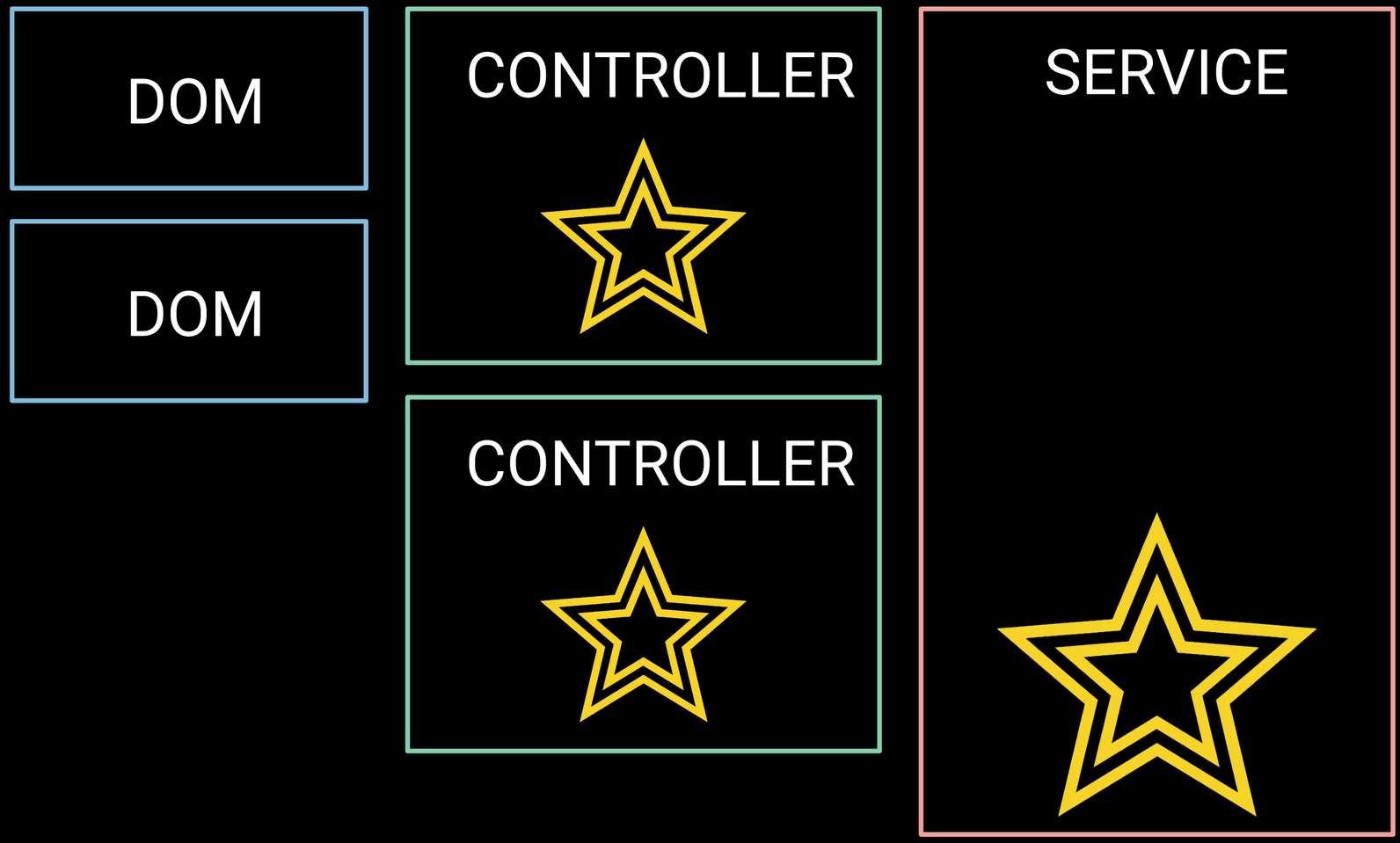
DOM

CONTROLLER

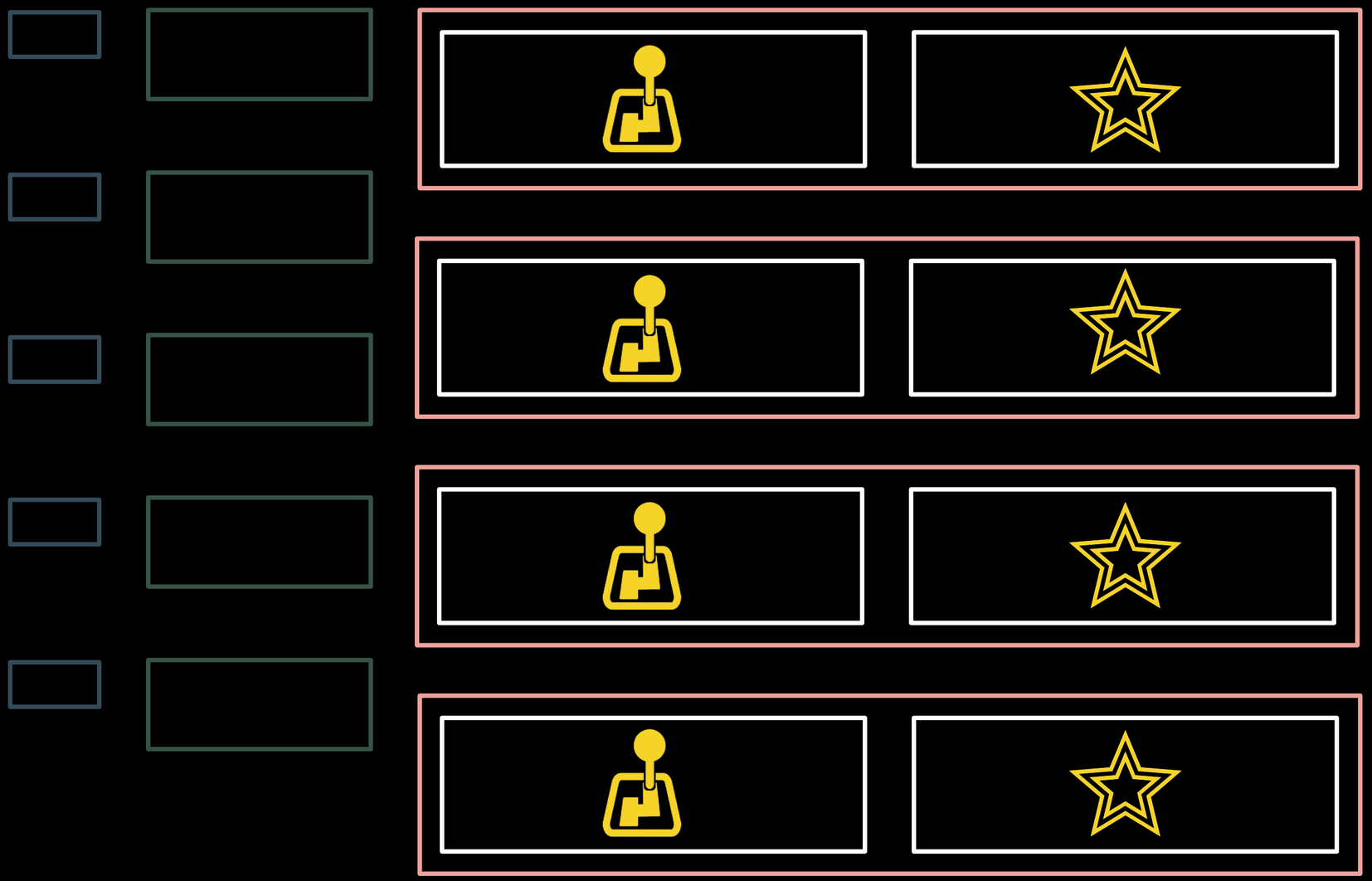
First Generation AngularJS Application



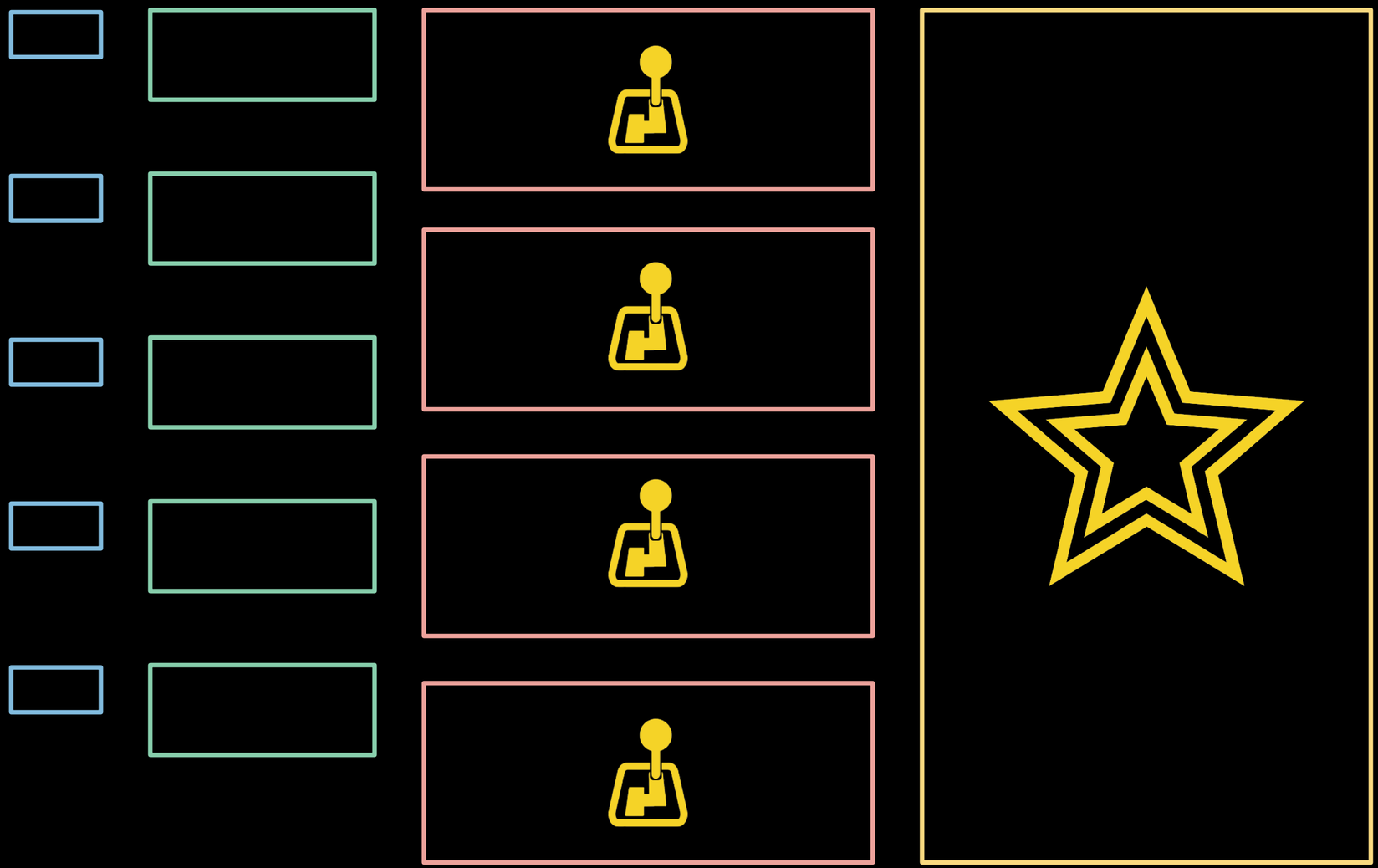
Second Generation AngularJS Application



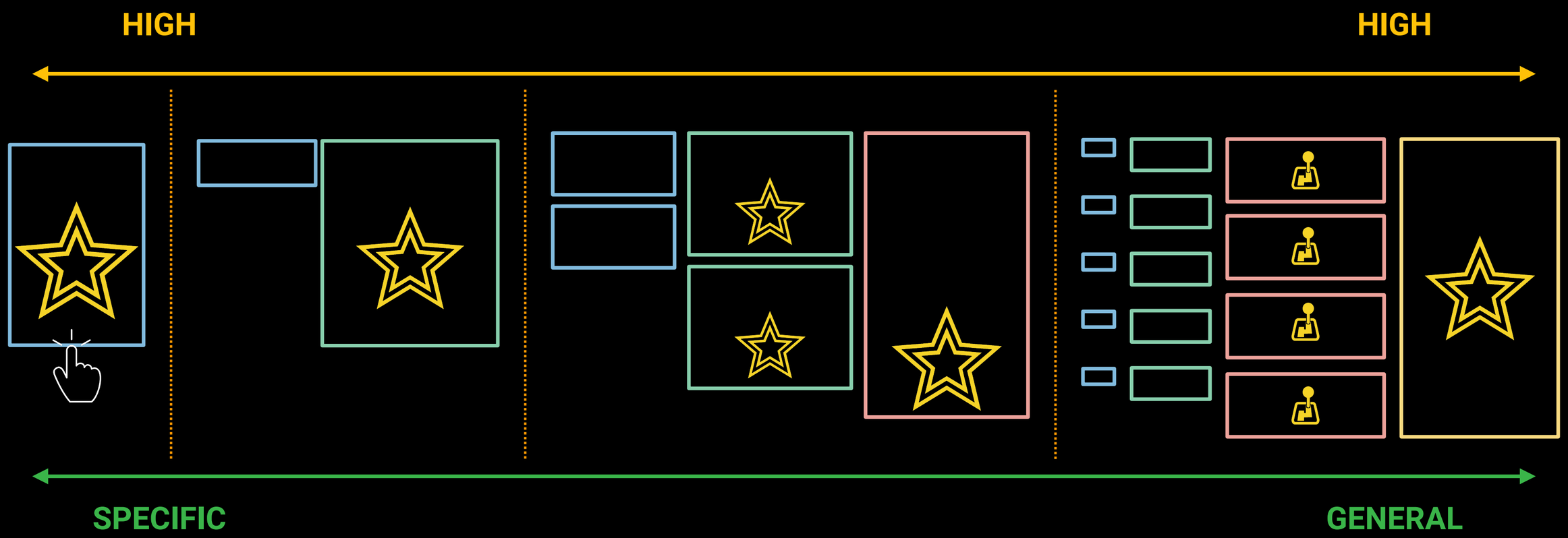
Second Generation AngularJS Application



Typical Stateful Service



Enter Redux


HIGH
HIGH
SPECIFIC
GENERAL

Good abstractions allow us to **write high-quality code once** and use it over and over

Specific details become reusable general details

Act Zero Big Idea

After 20 years of programming, I realized that I was essentially doing the **same four things** over and over.

Describing Things
Performing Actions
Making Decisions
Repeating via Iteration

Data Structures
Performing Actions
Making Decisions
Repeating via Iteration

Data Structures
Functions
Making Decisions
Repeating via Iteration

Data Structures
Functions
Conditionals
Repeating via Iteration

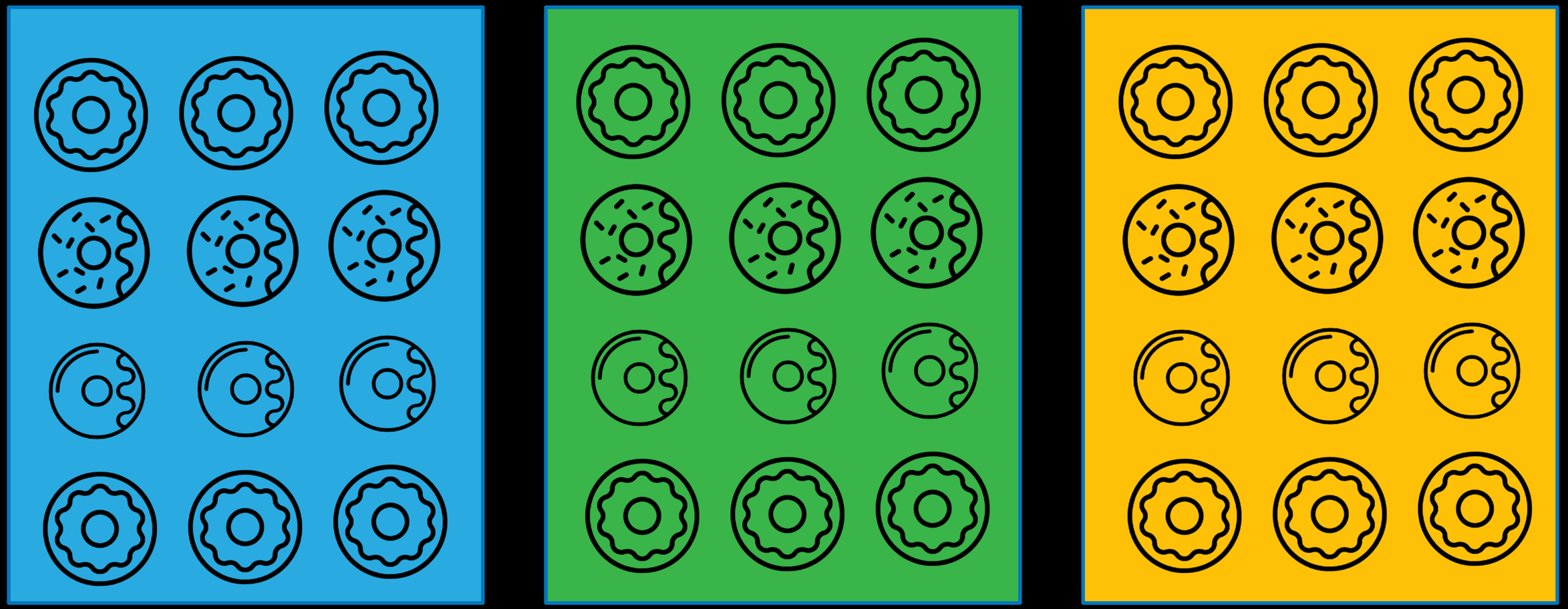
Data Structures
Functions
Conditionals
Iterators

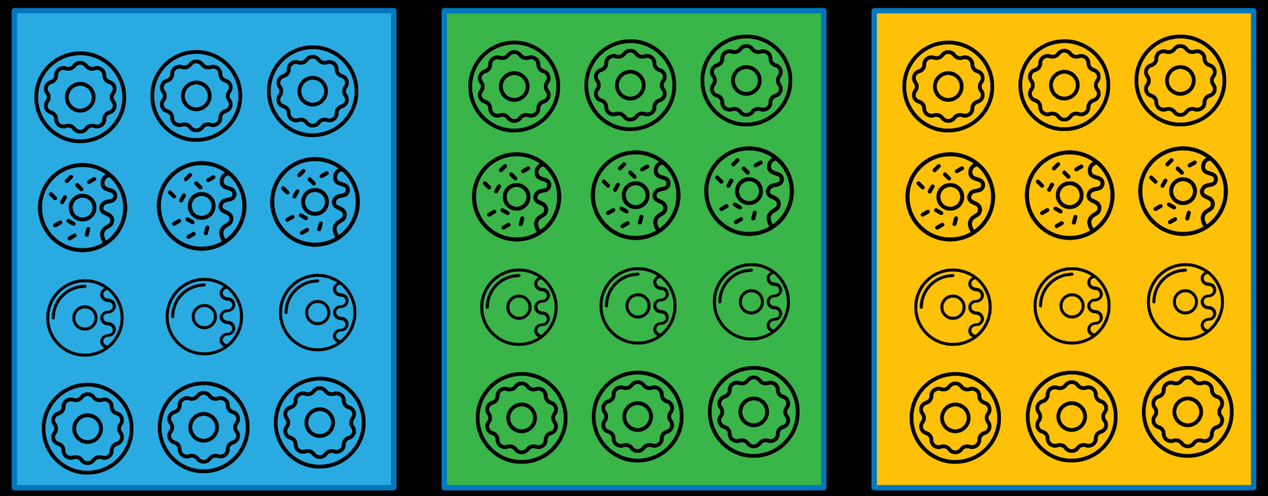
Nouns
Verbs
Conditionals
Iterators

I realized that I had been doing these things with ease since I was a **very small child**.

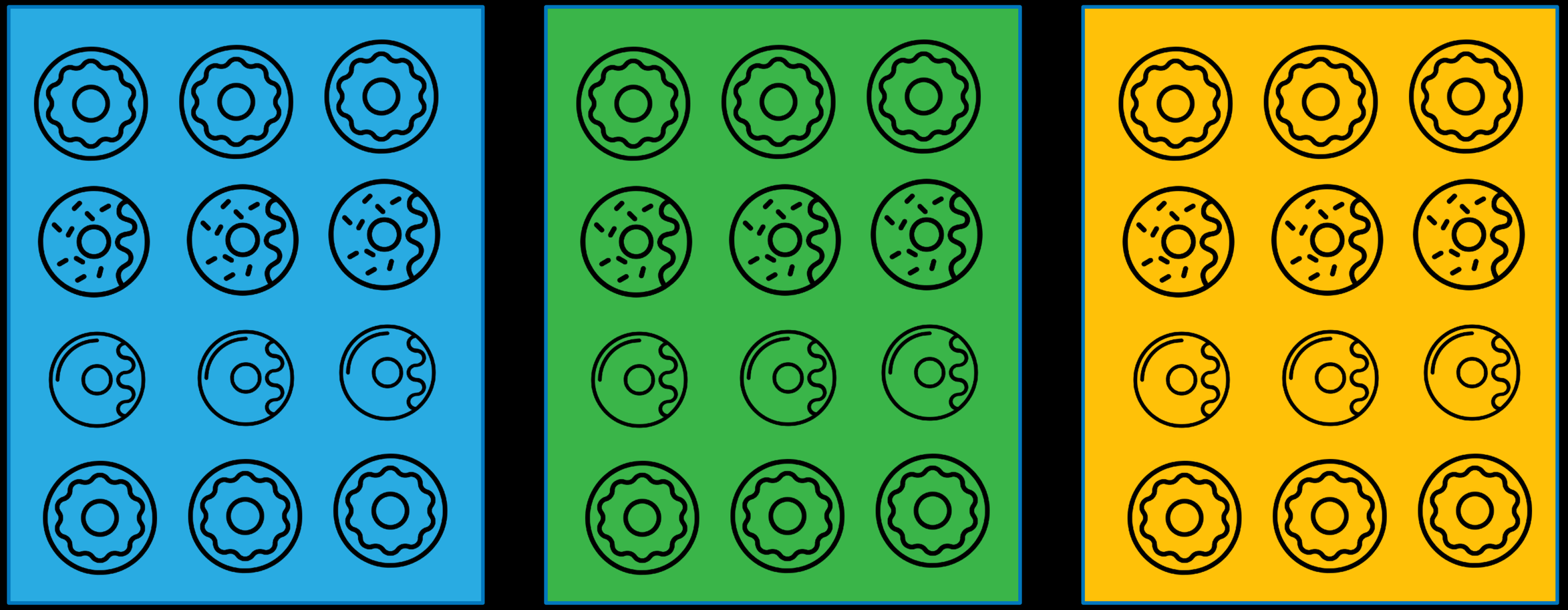
BONUS!

Thought Exercise





Do you see those trays of donuts?
Do you see the green tray?
On that tray, find all the donuts that have sprinkles on them.
Bring them back to me.



```
const selectedTray = trays.find(tray => tray.color === 'green');
const sprinkleDonuts = selectedTray.donuts.filter(donut => donut.sprinkles)
child.fetch(sprinkleDonuts);
```

Our goal is not to learn more but rather to **gain a deeper** understanding of what we already know.

Act One Mindset

What **limiting beliefs** exist around rapid prototyping?

Pen and Paper

Wireframes

Static Mocks

Interactive Mocks

High Fidelity Prototypes

Pen and Paper

Wireframes

Static Mocks

Interactive Mocks

High Fidelity Prototypes

Pen and Paper

Wireframes

Static Mocks

Interactive Mocks

Perceived High Level of Effort

High Fidelity Prototypes

Pen and Paper

Wireframes

Static Mocks

Interactive Mocks

CODE IS HARD!

High Fidelity Prototypes

"Just to be clear, this prototype must never, ever be put into production. We will throw this away and replace it with real code."

- Every Engineer Ever

Historically, prototypes are designated as disposable or "throw away" because they are not written to production standards.

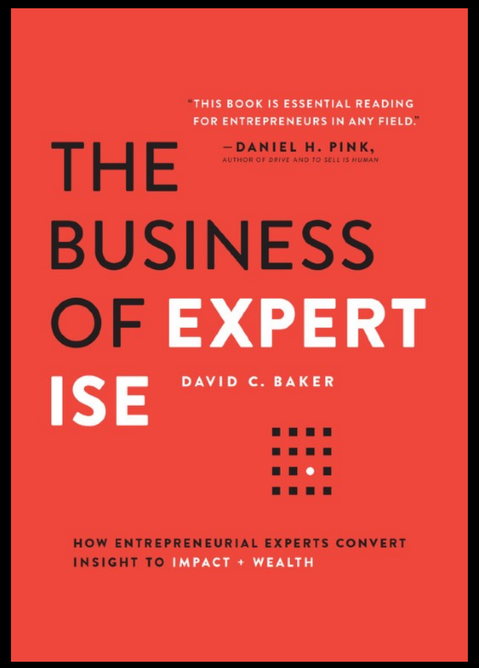
Prototypes should be **disposable** because they are **immutable** and **NOT** because they are **inferior**.

Immutable infrastructure is an approach to managing services and software deployments on IT resources wherein components are replaced rather than changed. An application or services is effectively redeployed each time any change occurs.

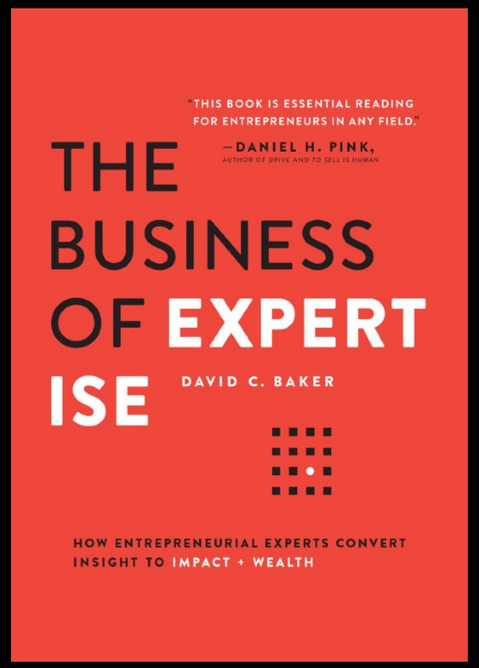
https://searchitoperations.techtarget.com/definition/immutable-infrastructure

Interlude Expertise

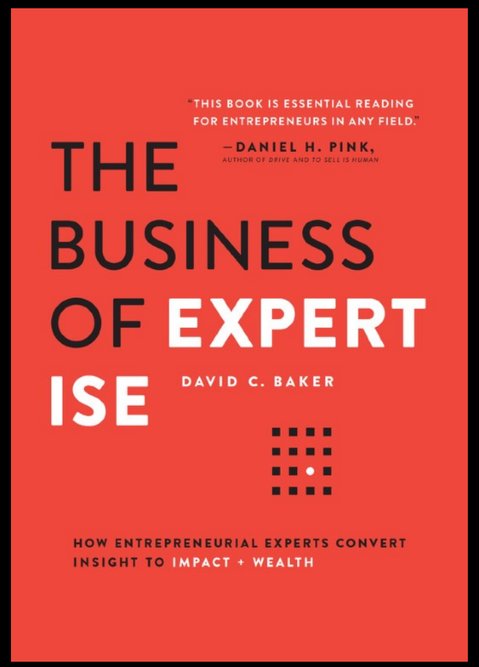
Experience and **Expertise**



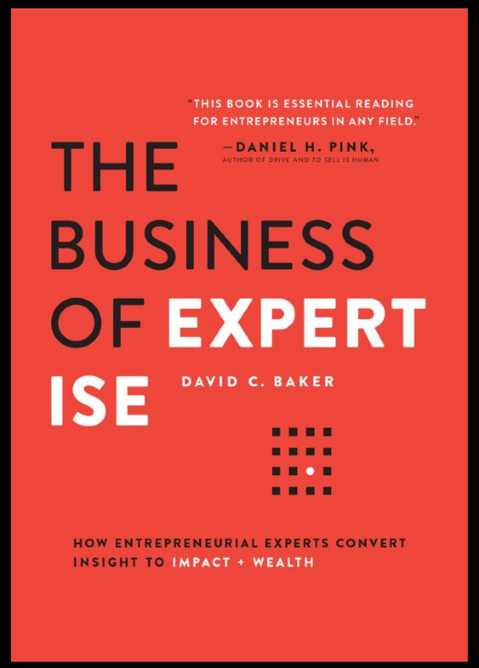
Intelligence is essentially pattern matching.



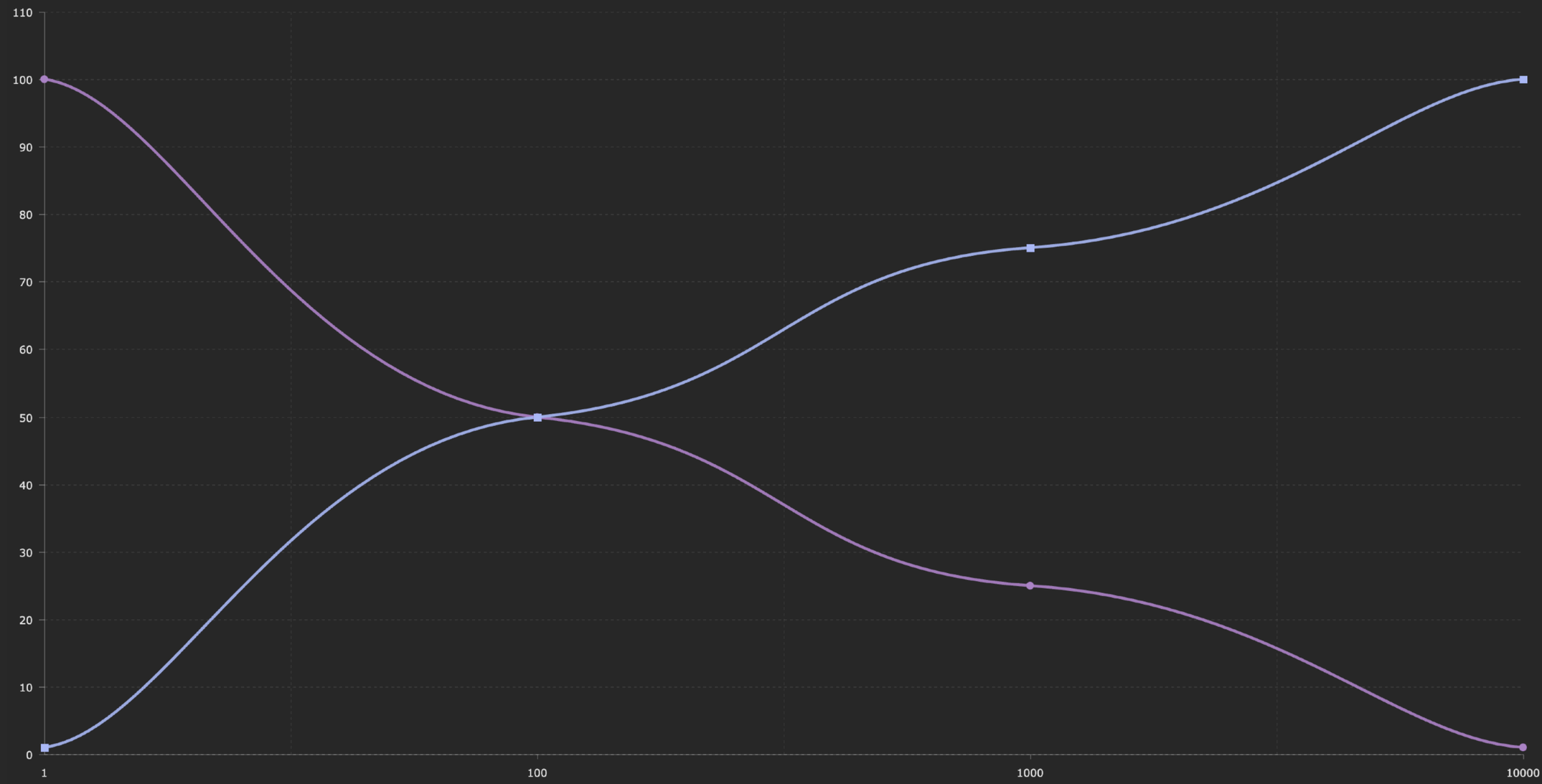
Pattern matching is possible when similar scenarios are presented.



When enough scenarios are presented, pattern matching turns into insights.



Expertise is the ability to convert these insights into valuable outcomes.



Skill ∝ Effort

Patterns are **Important**

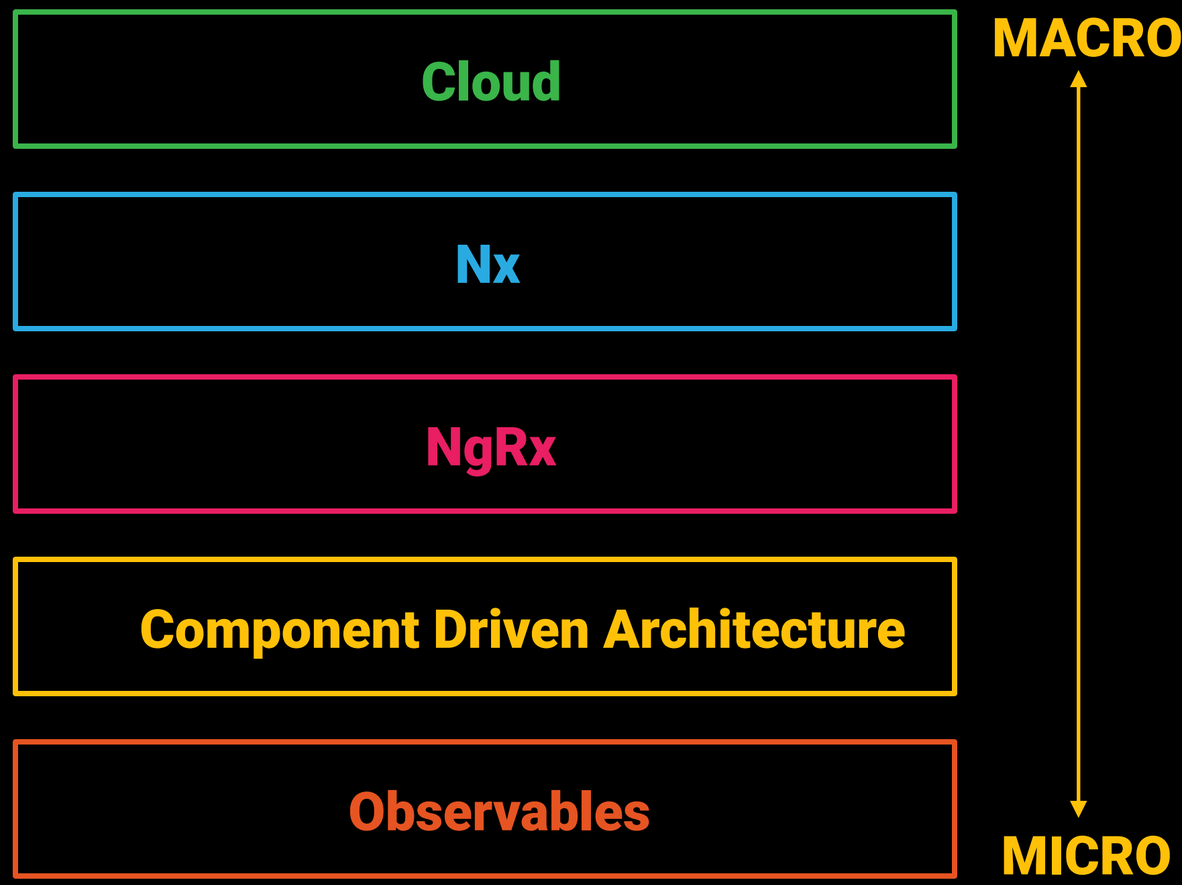
Act Two Architecture

The biggest problem in the development and maintenance of large-scale software systems is complexity — large systems are hard to understand.

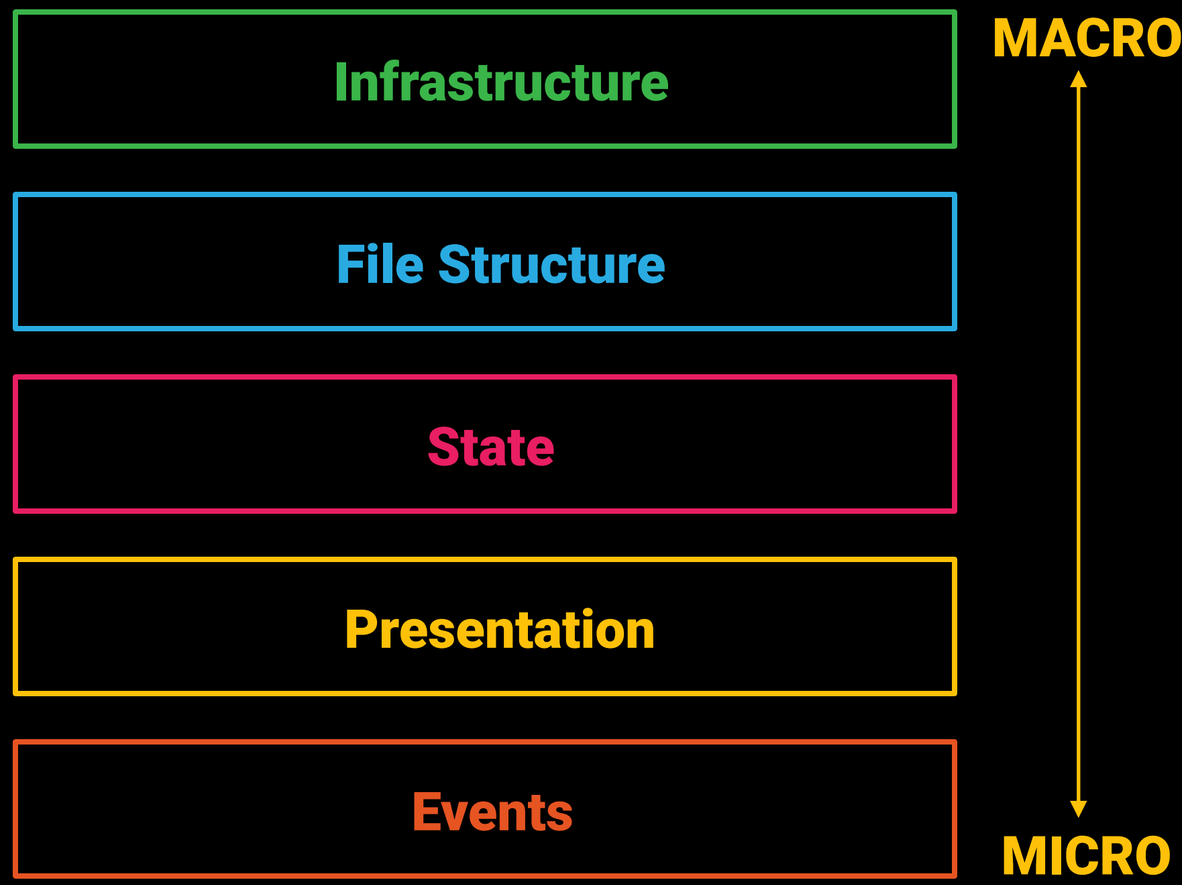
Out of the Tarpit - Ben Mosely Peter Marks

We believe that the major contributor to this complexity in many systems is the handling of state and the burden that this adds when trying to analyze and reason about the system. Other closely related contributors are code volume, and explicit concern with the flow of control through the system.

Out of the Tarpit - Ben Mosely Peter Marks

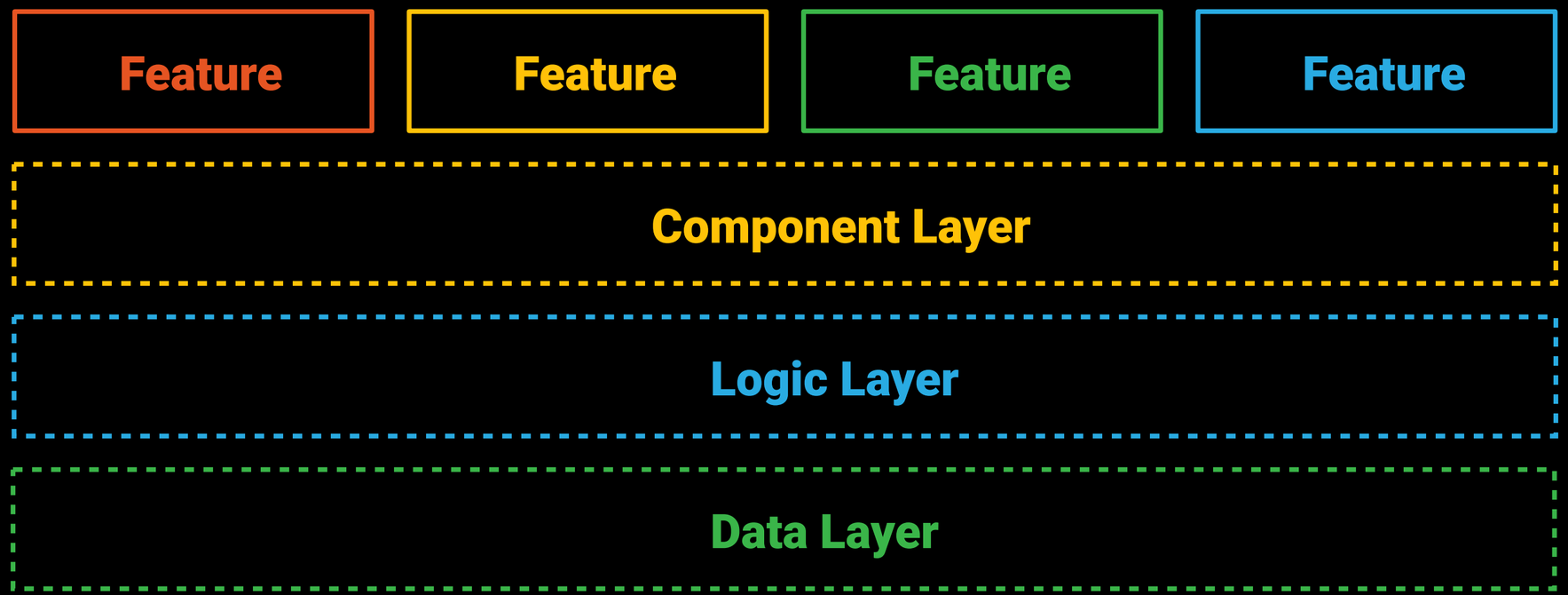


Managing Complexity



Managing Complexity

Vertical Development



Horizontal Development

Convention

Automation

Generation

Duplication

Elimination

Consolidation

Emerging Patterns

When we achieve clear **functional cohesion** from cleanly **abstracted layers**, natural **patterns of convention** will emerge.

Fine-grained functionality reduces code variance between domain models. Performing an action on one model is almost identical to the same action on any other model.

```
const _coursesReducer = createReducer(
  initialCoursesState,
  // Load courses
  on(CoursesActions.loadCourses, state => ({ ...state, loaded: false, error: null })),
  on(CoursesActions.loadCoursesSuccess, (state, { courses }) =>
    coursesAdapter.setAll(courses, { ...state, loaded: true })
  ),
  on(CoursesActions.loadCoursesFailure, onFailure),
  // Add course
  on(CoursesActions.createCourseSuccess, (state, { course }) =>
    coursesAdapter.addOne(course, state)
  ),
  on(CoursesActions.createCourseFailure, onFailure),
  // Update course
  on(CoursesActions.updateCourseSuccess, (state, { course }) =>
    coursesAdapter.updateOne({ id: course.id, changes: course }, state)
  ),
  on(CoursesActions.updateCourseFailure, onFailure),
  // Delete course
  on(CoursesActions.deleteCourseSuccess, (state, { course }) =>
    coursesAdapter.removeOne(course.id, state)
  ),
  on(CoursesActions.deleteCourseFailure, onFailure),
);
```

```
const _lessonsReducer = createReducer(
  initialLessonsState,
  // Load lessons
  on(LessonsActions.loadLessons, state => ({ ...state, loaded: false, error: null })),
  on(LessonsActions.loadLessonsSuccess, (state, { lessons }) =>
    lessonsAdapter.setAll(lessons, { ...state, loaded: true })
  ),
  on(LessonsActions.loadLessonsFailure, onFailure),
  // Add lesson
  on(LessonsActions.createLessonSuccess, (state, { lesson }) =>
    lessonsAdapter.addOne(lesson, state)
  ),
  on(LessonsActions.createLessonFailure, onFailure),
  // Update lesson
  on(LessonsActions.updateLessonSuccess, (state, { lesson }) =>
    lessonsAdapter.updateOne({ id: lesson.id, changes: lesson }, state)
  ),
  on(LessonsActions.updateLessonFailure, onFailure),
  // Delete lesson
  on(LessonsActions.deleteLessonSuccess, (state, { lesson }) =>
    lessonsAdapter.removeOne(lesson.id, state)
  ),
  on(LessonsActions.deleteLessonFailure, onFailure),
);
```

```ts
const _coursesReducer = createReducer(
  initialCoursesState,
  // Load courses
  on(CoursesActions.loadCourses, state => ({ ...state, loaded: false,
error: null })),
  on(CoursesActions.loadCoursesSuccess, (state, { courses }) =>
    coursesAdapter.setAll(courses, { ...state, loaded: true })
  ),
  on(CoursesActions.loadCoursesFailure, onFailure),
  // Add course
  on(CoursesActions.createCourseSuccess, (state, { course }) =>
    coursesAdapter.addOne(course, state)
  ),
  on(CoursesActions.createCourseFailure, onFailure),
  // Update course
  on(CoursesActions.updateCourseSuccess, (state, { course }) =>
    coursesAdapter.updateOne({ id: course.id, changes: course }, state)
  ),
  on(CoursesActions.updateCourseFailure, onFailure),
  // Delete course
  on(CoursesActions.deleteCourseSuccess, (state, { course }) =>
    coursesAdapter.removeOne(course.id, state)
  ),
  on(CoursesActions.deleteCourseFailure, onFailure),
);
```

```ts
const _lessonsReducer = createReducer(
  initialLessonsState,
  // Load lessons
  on(LessonsActions.loadLessons, state => ({ ...state, loaded: false, error:
null })),
  on(LessonsActions.loadLessonsSuccess, (state, { lessons }) =>
    lessonsAdapter.setAll(lessons, { ...state, loaded: true })
  ),
  on(LessonsActions.loadLessonsFailure, onFailure),
  // Add lesson
  on(LessonsActions.createLessonSuccess, (state, { lesson }) =>
    lessonsAdapter.addOne(lesson, state)
  ),
  on(LessonsActions.createLessonFailure, onFailure),
  // Update lesson
  on(LessonsActions.updateLessonSuccess, (state, { lesson }) =>
    lessonsAdapter.updateOne({ id: lesson.id, changes: lesson }, state)
  ),
  on(LessonsActions.updateLessonFailure, onFailure),
  // Delete lesson
  on(LessonsActions.deleteLessonSuccess, (state, { lesson }) =>
    lessonsAdapter.removeOne(lesson.id, state)
  ),
  on(LessonsActions.deleteLessonFailure, onFailure),
);
```

What is different?

This is why
good architecture
is so powerful.

Act Three Tactical

Application **Application**

Do you have more than
one application?

Do they share any **domain models**?

As a result, do they share any **business logic**?

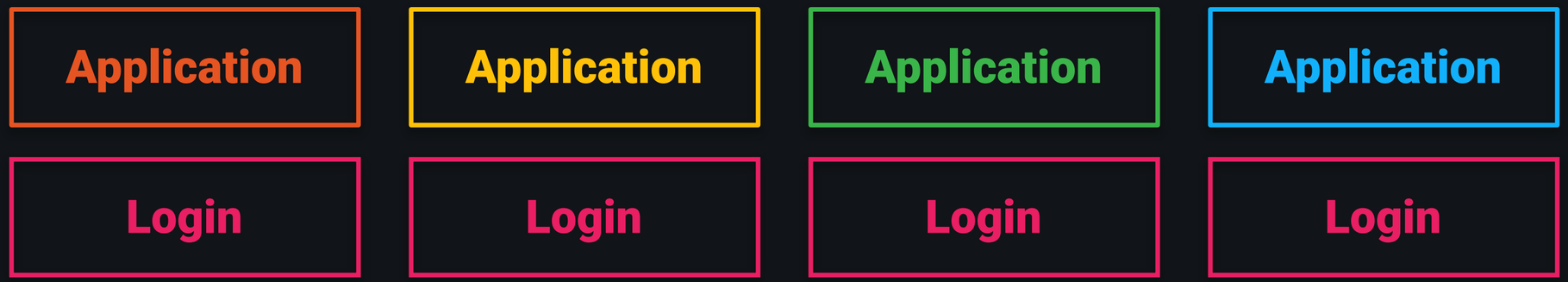
As a result, do they share any
common functionality?

The **login functionality** is *probably* not going to change from one application to the next.

The **HTTP service functionality** is *probably* not going to change from one application to the next.

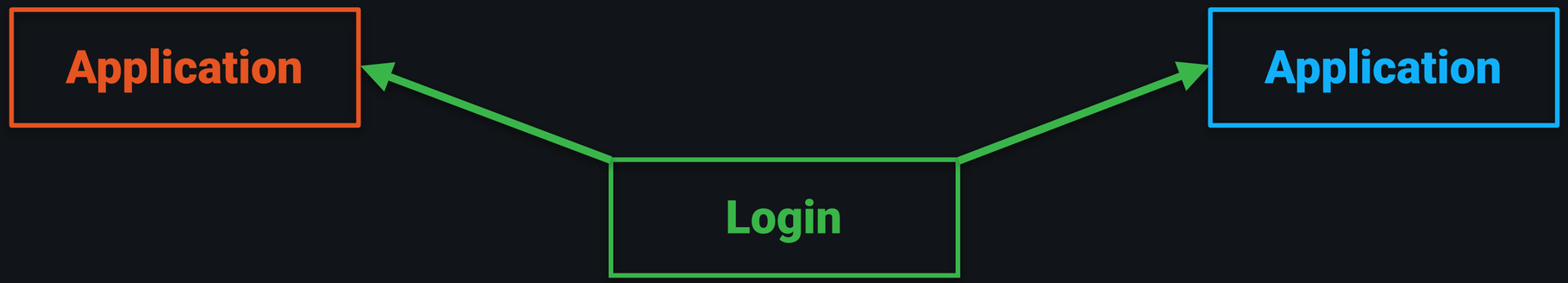
It is also *reasonable* to assume that your applications may share a **common set of UI components** for the sake of consistency.

How do you manage these **shared dependencies** without duplicating code?

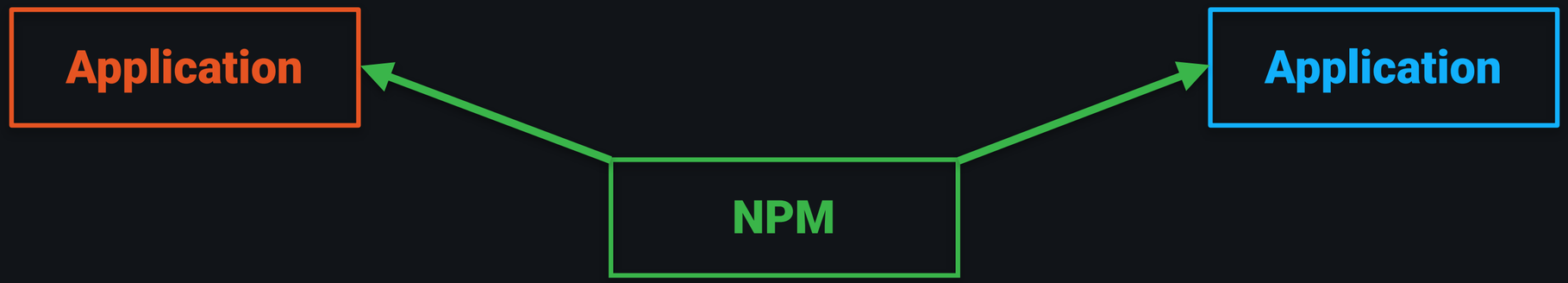


Let's agree that **code duplication** is **not** a cost-effective, scalable option.

With that said, you have **two viable options** that you can adopt.

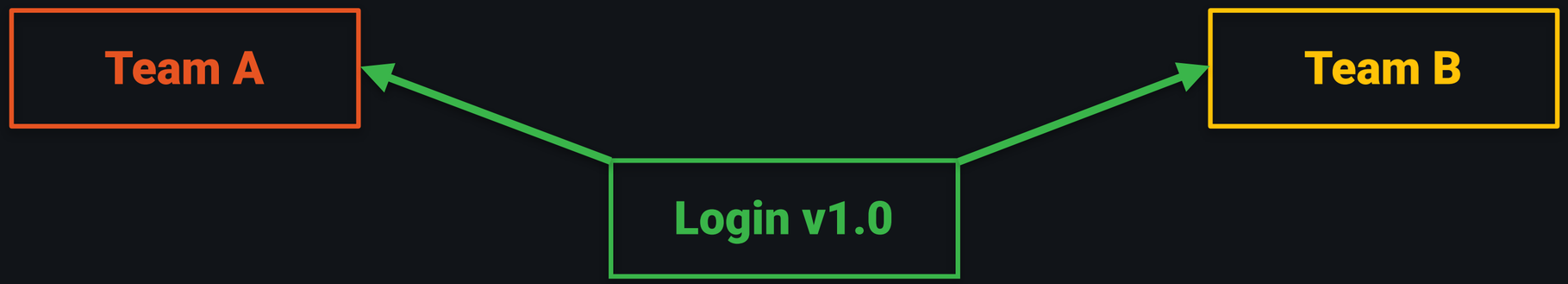


Extract common functionality into separate packages and publish into a public or private registry.

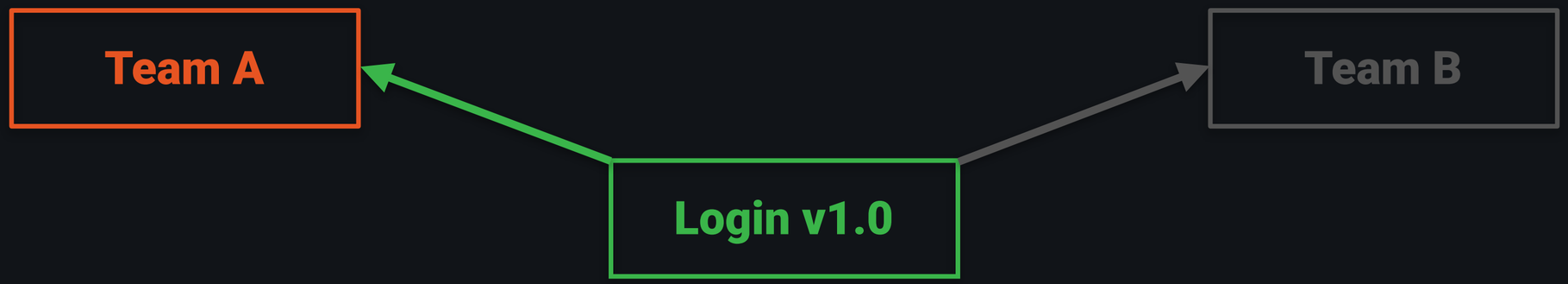


This is basically how public frameworks and component libraries work.

This is the only viable option when you do not have the luxury of operating within a **single organization**.



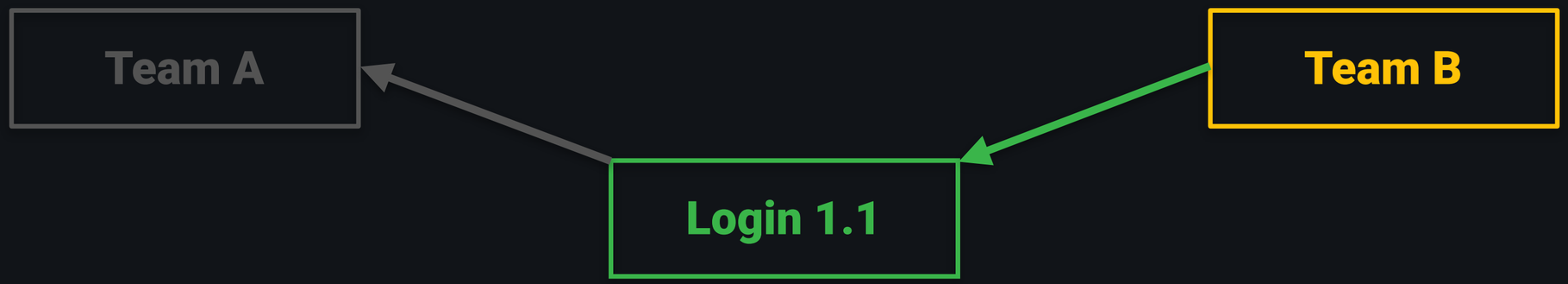
For example...



If **Team A** needs a change...

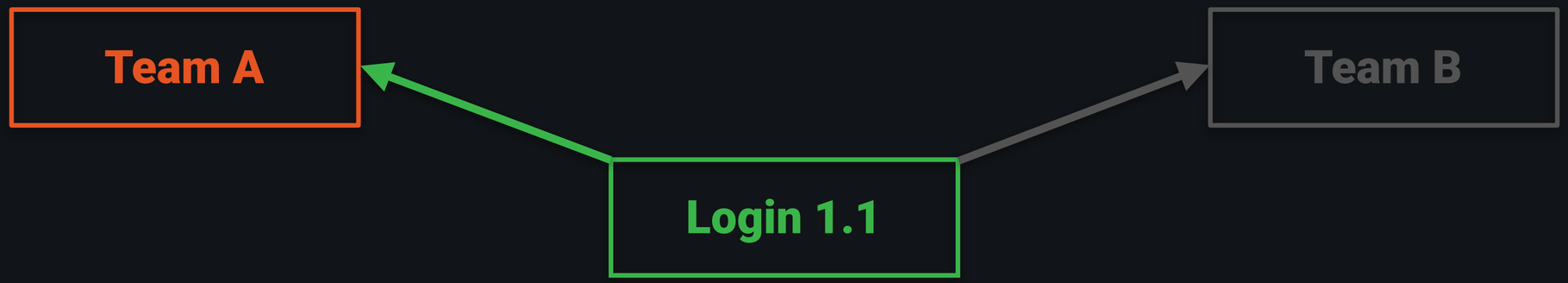
Team A notifies Team B...

Team B will eventually make the change...



Team B will publish the change...

Team B will notify **Team A** of the new version...



Team A will then update their dependencies to use the new version

What happens if **Team B** does not share the same sense of urgency as **Team A**? What happens if something breaks?

The challenge with this is the **asynchronous** and **unpredictable** manner in which changes are made available to consuming projects.

This is also a *great* way to break everything which is why **JavaScript Fatigue™** is a real thing.

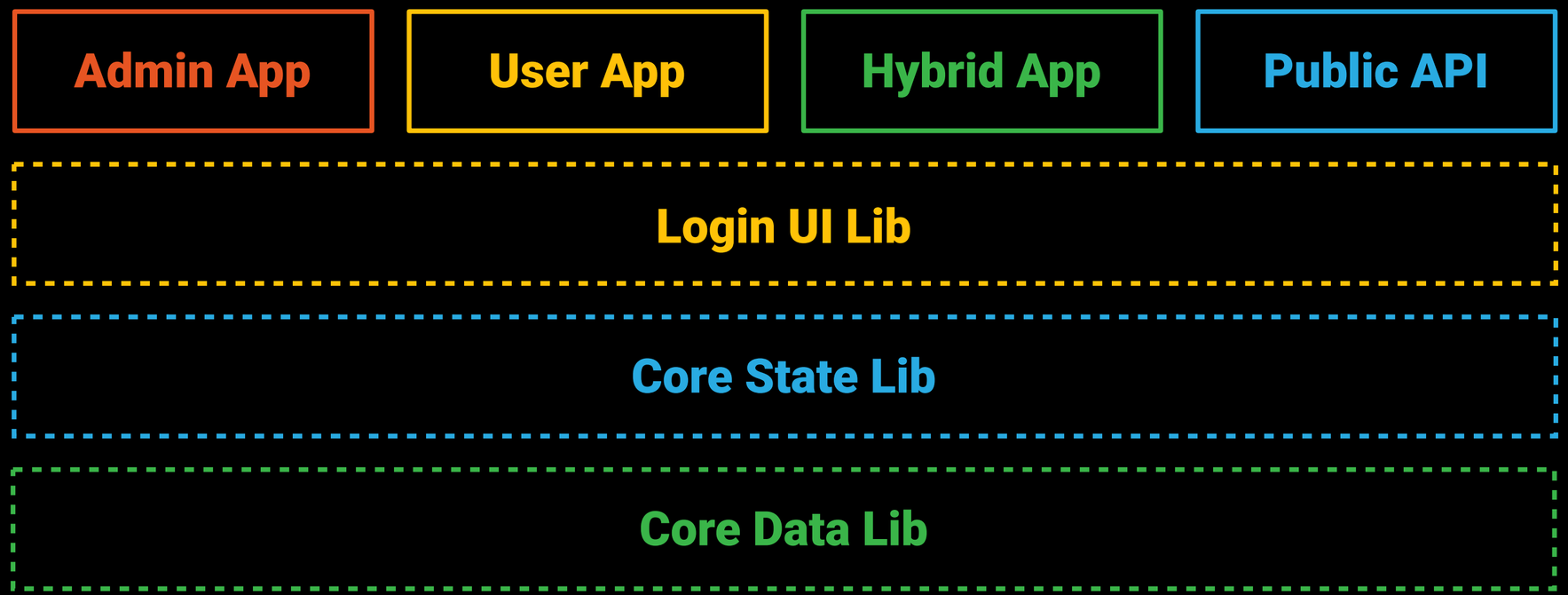
If you have the luxury of operating within a **single organization**, you can circumvent this entirely.

You can put all of your code into a **single mono repository** with all of its internal dependencies **neatly organized for proximity**.

In fact, this is what **Google** does internally.

Nrwl Extensions

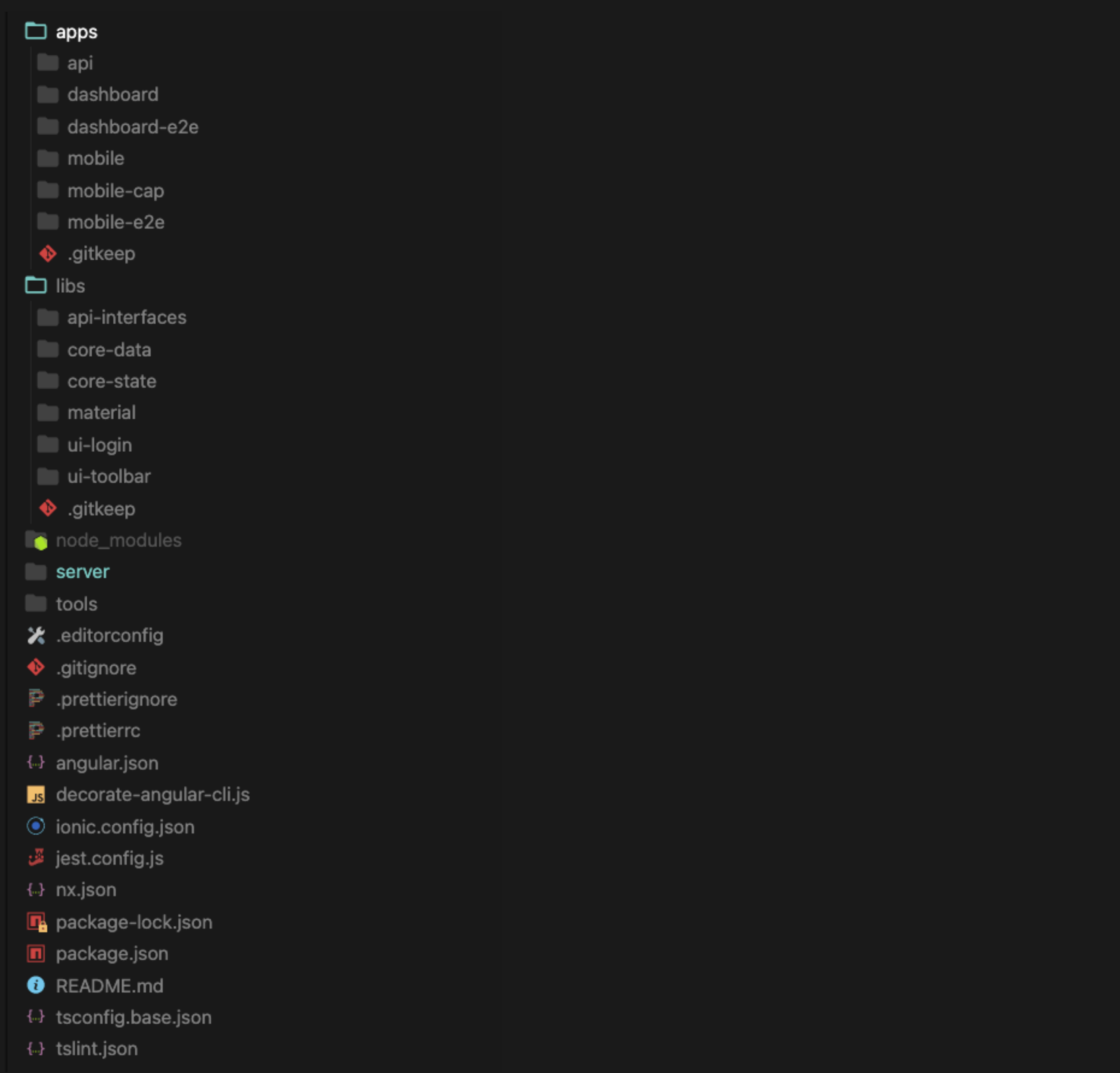
This approach is what two
Ex-Googlers used to create **Nx**



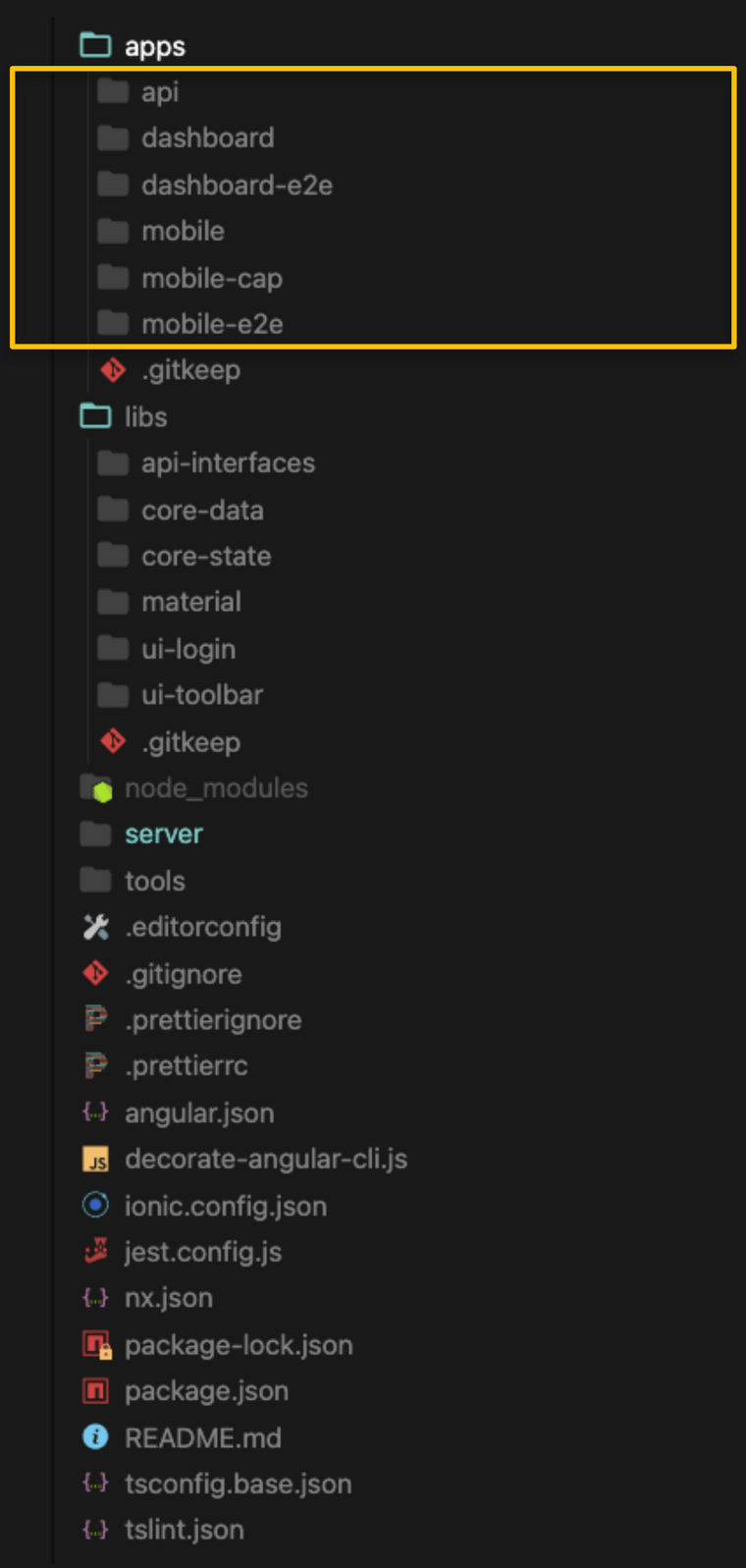
Monorepo Approach

Nx allows you to keep **N number of applications** in close proximity while allowing them to **share dependencies** without having to incur the **overhead of a highly inefficient, asynchronous workflow**.

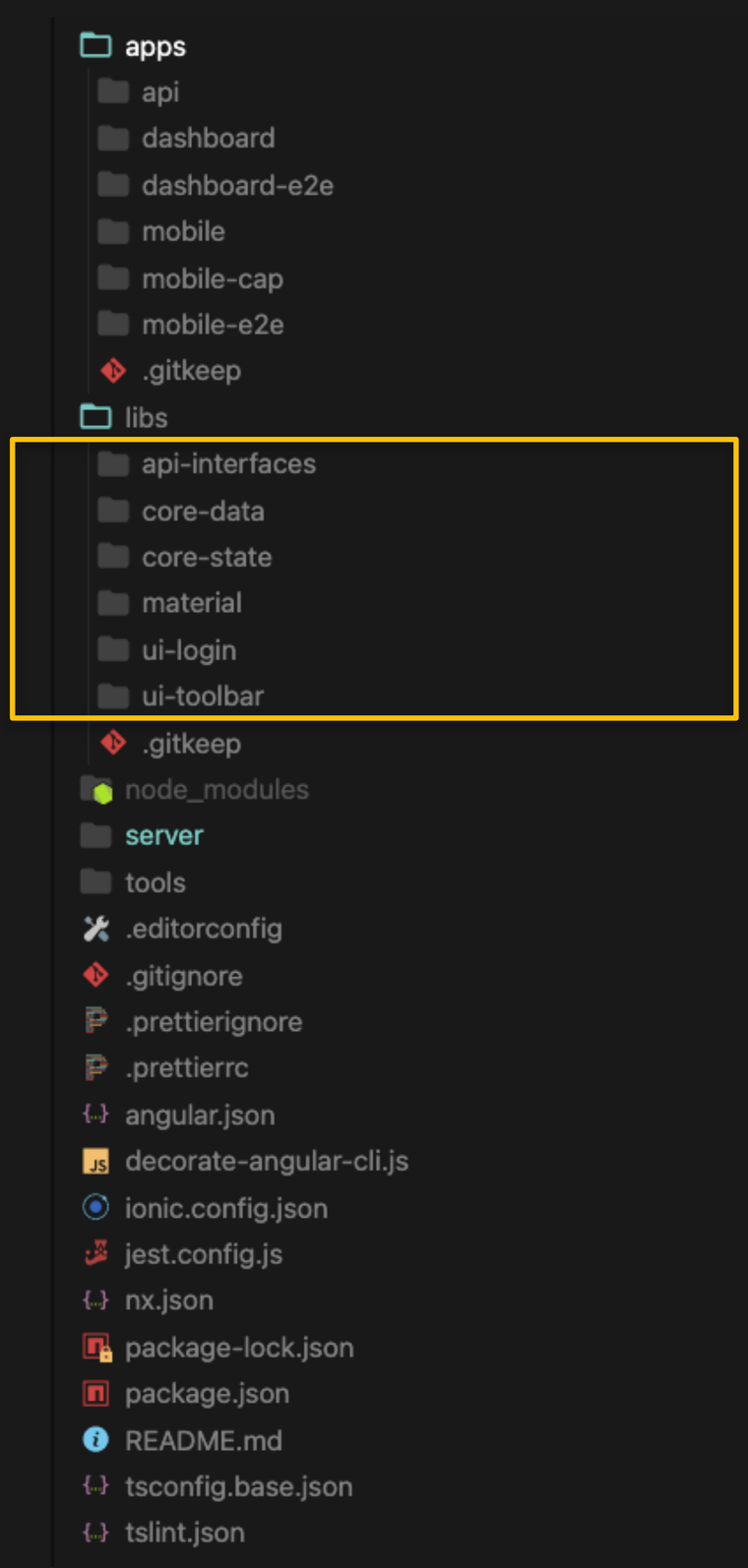
For Example


apps
api
dashboard
dashboard-e2e
mobile
mobile-cap
mobile-e2e
.gitkeep
libs
api-interfaces
core-data
core-state
material
ui-login
ui-toolbar
.gitkeep
node_modules
server
tools
.editorconfig
.gitignore
.prettierignore
.prettierrc
angular.json
decorate-angular-cli.js
ionic.config.json
jest.config.js
nx.json
package-lock.json
package.json
README.md
tsconfig.base.json
tslint.json

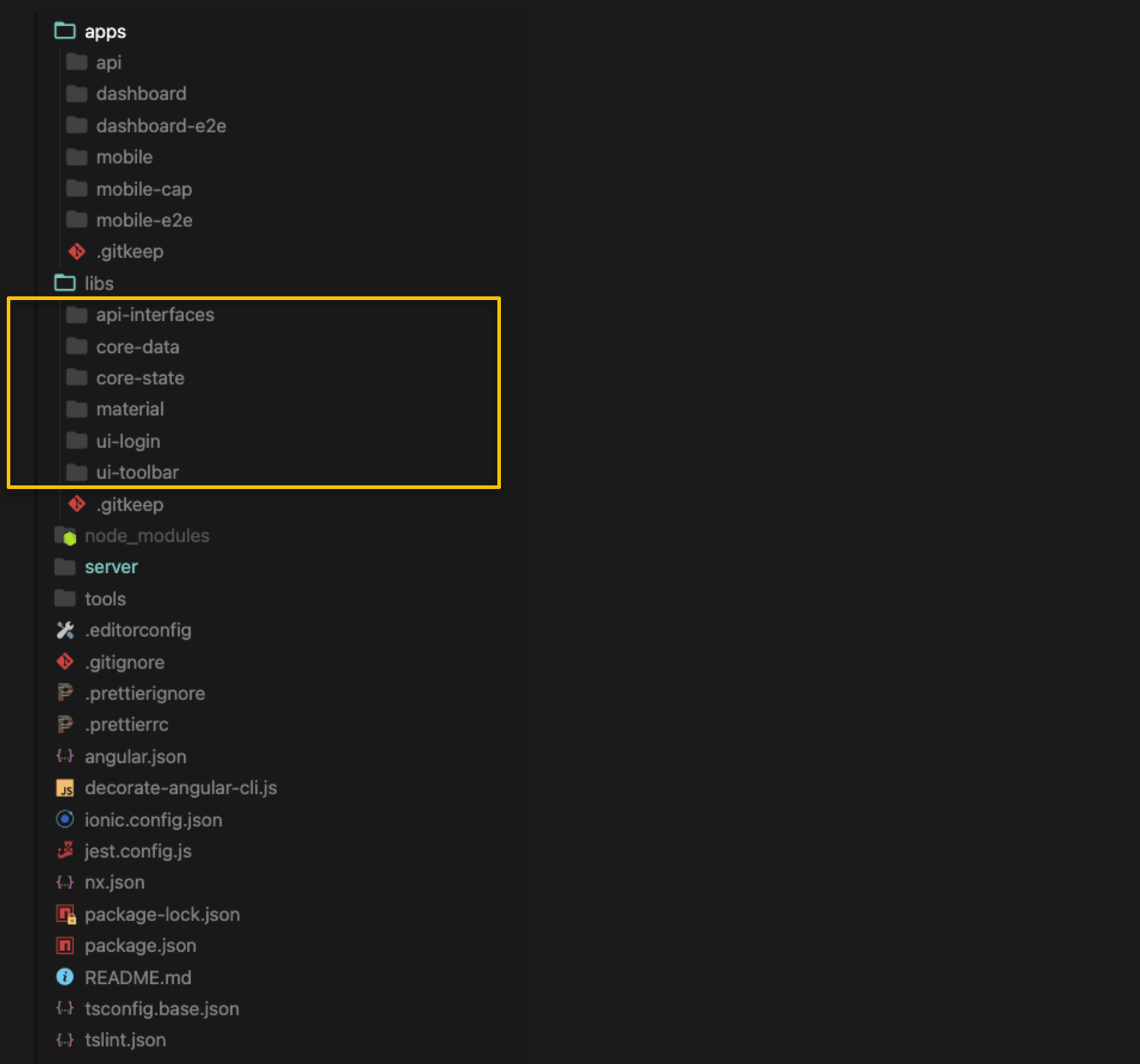
Six Apps


apps
api
dashboard
dashboard-e2e
mobile
mobile-cap
mobile-e2e
.gitkeep
libs
api-interfaces
core-data
core-state
material
ui-login
ui-toolbar
.gitkeep
node_modules
server
tools
.editorconfig
.gitignore
.prettierignore
.prettierrc
angular.json
decorate-angular-cli.js
ionic.config.json
jest.config.js
nx.json
package-lock.json
package.json
README.md
tsconfig.base.json
tslint.json

Share Six Common Libraries



Write Once
Save Five



This is very, very cost effective.


AND NOW
LET'S GO!

Bonus! Axis of Evil™

Architecture is the tension between **coupling** and **cohesion**.

- Neal Ford

We cannot effectively discuss the **any design pattern** without first discussing **coupling**, **abstraction**, and **cohesion**.

TLDR

Tightly coupled code is highly dependent code
As dependencies increase, future effort also increases
As dependencies increase, future options decreases
Coupling is solved via abstraction
Abstraction is optimization via organization
High cohesion is the result of optimal organization
Cohesion should create logical boundaries within a system with clear and precise semantics

Concept:

Coupling

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {
  switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }

  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Tightly Coupled Code

```
const testService = new RefactorService();
const testWidget = { id: 100, name: '', price: 100, description: ''};
const testWidgets = [{ id: 100, name: '', price: 200, description: ''}];
testService.widgets = testWidgets;

testService.mode = 'create';
testService.reCalculateTotal(testWidget);

testService.mode = 'update';
testService.reCalculateTotal(testWidget);

testService.mode = 'delete';
testService.reCalculateTotal(testWidget);
```

What is the result?

```
const testService = new RefactorService();
const testWidget = { id: 100, name: '', price: 100, description: ''};
const testWidgets = [{ id: 100, name: '', price: 200, description: ''}];
testService.widgets = testWidgets;

testService.mode = 'create';
testService.reCalculateTotal(testWidget);

testService.mode = 'update';
testService.reCalculateTotal(testWidget);

testService.mode = 'delete';
testService.reCalculateTotal(testWidget);
```

It is impossible to know!

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {

  switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }


  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Axis of Evil™ Hidden State

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {

   switch (this.mode) {
    case 'create':
      const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
      this.widgets = [...this.widgets, newWidget];
      break;
    case 'update':
      this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
      break;
    case 'delete':
      this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
      break;
    default:
      break;
  }


   this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Axis of Evil™ Nested Logic

```
price;
mode;
widgets: Widget[];

reCalculateTotal(widget: Widget) {
  switch (this.mode) {
   case 'create':
     const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
     this.widgets = [...this.widgets, newWidget];
     break;
   case 'update':
     this.widgets = this.widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
     break;
   case 'delete':
     this.widgets = this.widgets.filter(wdgt => widget.id !== wdgt.id);
     break;
   default:
     break;
  }

  this.price = this.widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Axis of Evil™ SRP Violation

What is the **solution**?

Break the **coupling**!

Extract to **parameter** 🎉

Extract to **method** 🎉

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

```
refactoredCalculateTotal(mode, widgets, newWidget) {
  this.widgets = this.updateWidgets(mode, widgets, newWidget);
  this.price = this.getTotalPrice(this.widgets);
}
updateWidgets(mode, widgets, newWidget) {
  switch (mode) {
    case 'create':
      return this.addWidget(widgets, newWidget);
    case 'update':
      return this.updateWidget(widgets, newWidget);
    case 'delete':
      return this.deleteWidget(widgets, newWidget);
    default:
      return widgets;
  }
}
addWidget(widgets, widget) {
  const newWidget = Object.assign({}, widget, {id: UUID.UUID()});
  return [...widgets, newWidget];
}
updateWidget(widgets, widget) {
  return widgets.map(wdgt => (widget.id === wdgt.id) ? Object.assign({}, widget) : wdgt);
}
deleteWidget(widgets, widget) {
  return widgets.filter(wdgt => widget.id !== wdgt.id);
}
getTotalPrice(widgets) {
  return widgets.reduce((acc, curr) => acc + curr.price, 0);
}
```

Bonus! Complexity

Make it work.
Make it right.
Make it fast.

- Kent Block

Make it work.

Make it known.

Make it right.

Make it fast.

- Lukas Ruebbelke

Code should be **fine**
grained

Code should **do one**
thing

Code should be **self**
documenting

Favor **pure, immutable**
functions

Abstractions should reduce complexity

Abstractions should reduce coupling

Abstractions should increase cohesion

Abstractions should increase portability

Refactor through promotion

Composition over inheritance

Do not confuse
convention for
repetition

Well-structured code
will naturally have a
larger surface area

Team rules for managing complexity

Be mindful over the
limitations of your
entire team and
optimize around that

Favor best practices
over introducing idioms
however clever they
may be

Consistency is better
than **righteousness**

Follow the style guide until it doesn't make sense for your situation

Tactical rules for managing complexity

Eliminate **hidden state** in functions

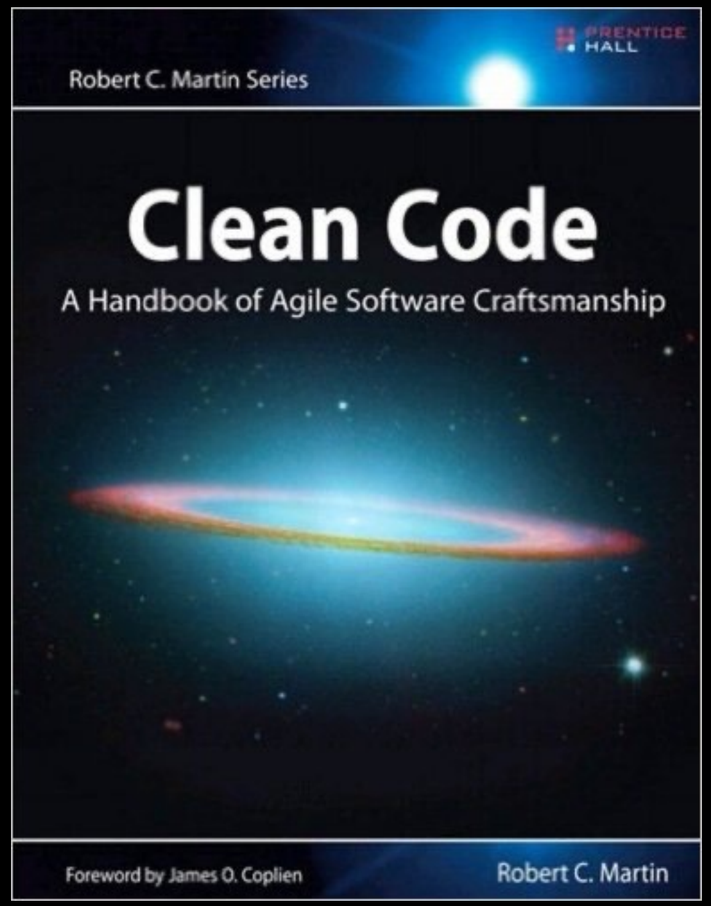
Eliminate **nested logic** in functions

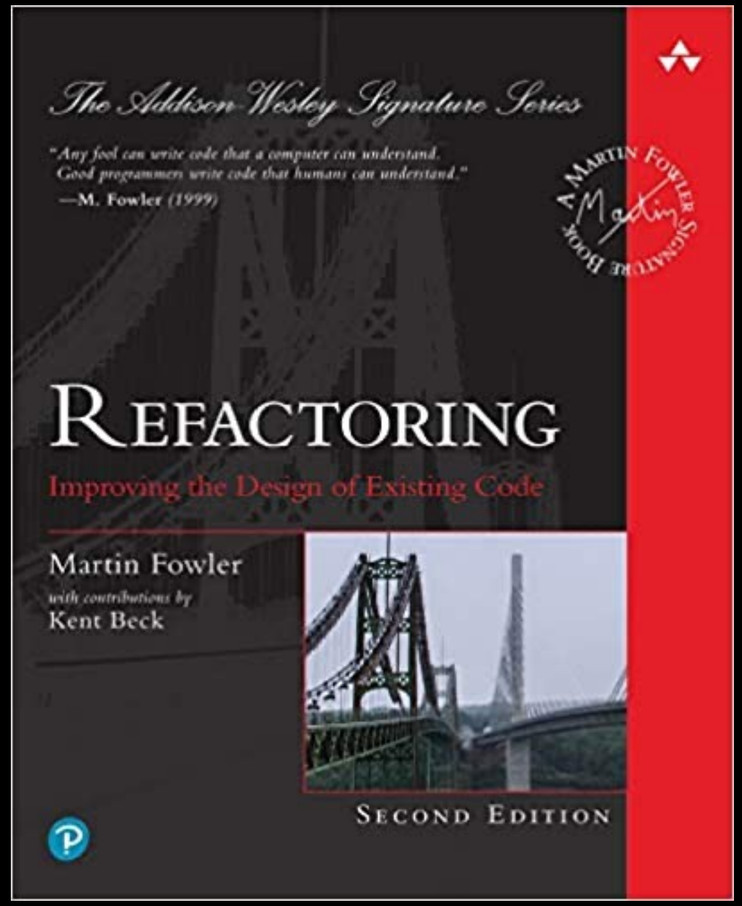
Do not break the **Single Responsibility** Principle

Extracting to a method is one of the most effective refactoring strategies available

If you need to **clarify your code with comments** then it is *probably* too complex

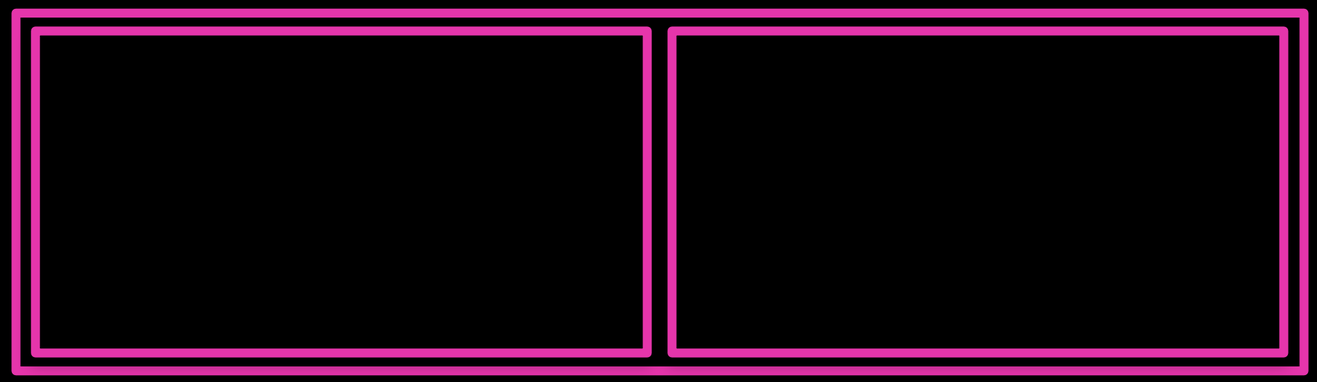
It is *impossible* to write **good tests** for **bad code**


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

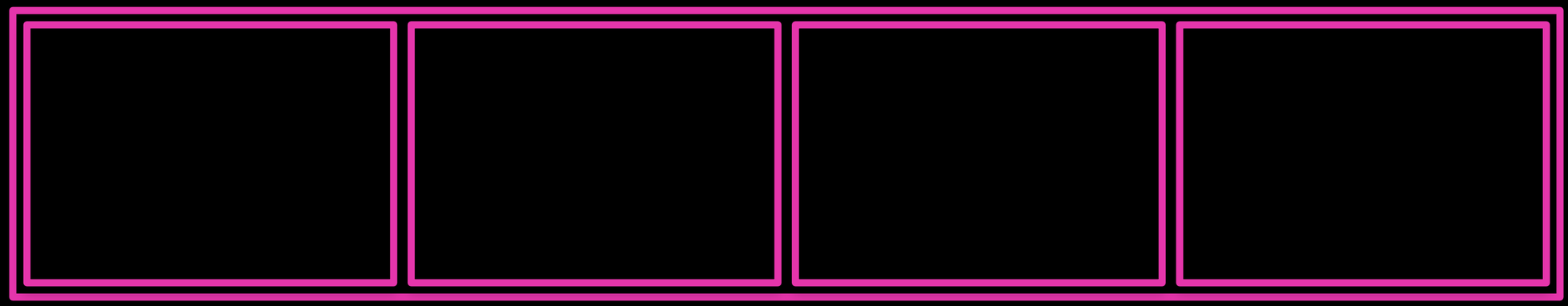

The Addison Wesley Signature Series
"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
—M. Fowler (1999)
A MARTIN FOWLER SIGNATURE BOOK
REFACTORING
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
SECOND EDITION

Bonus! Enterprise

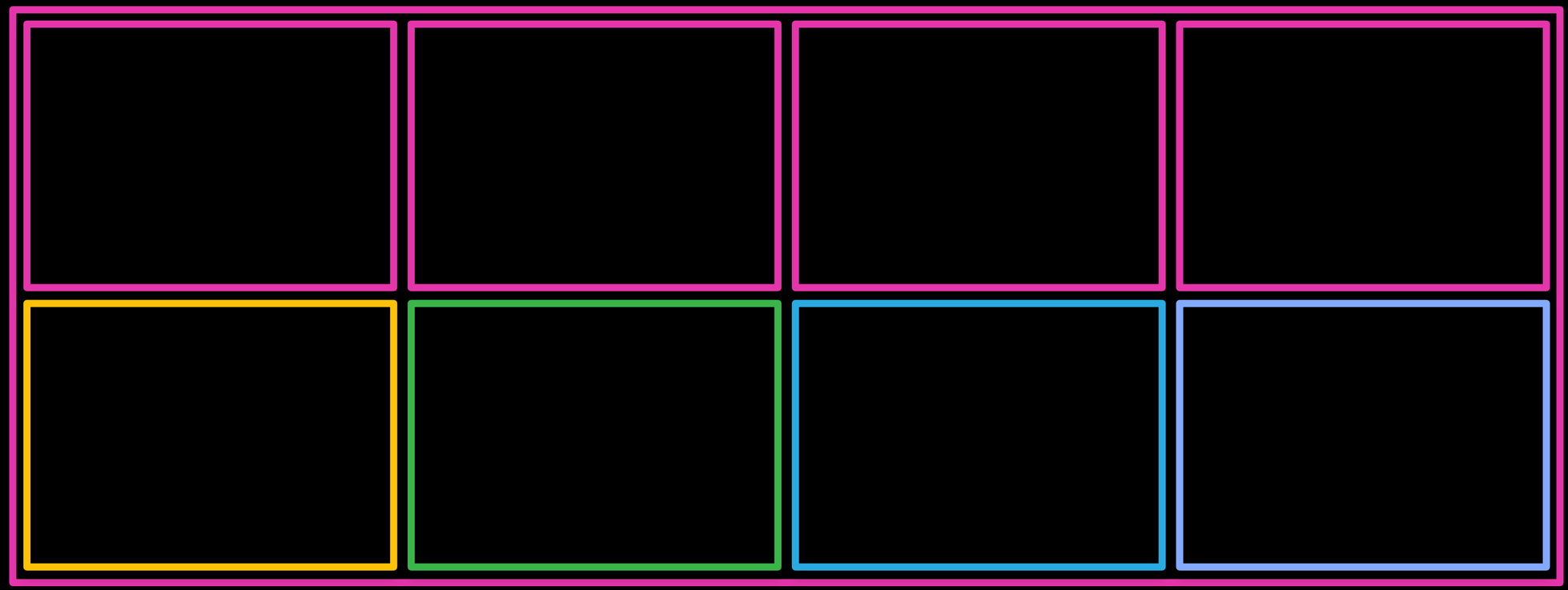
Application Inception



Application Complexity



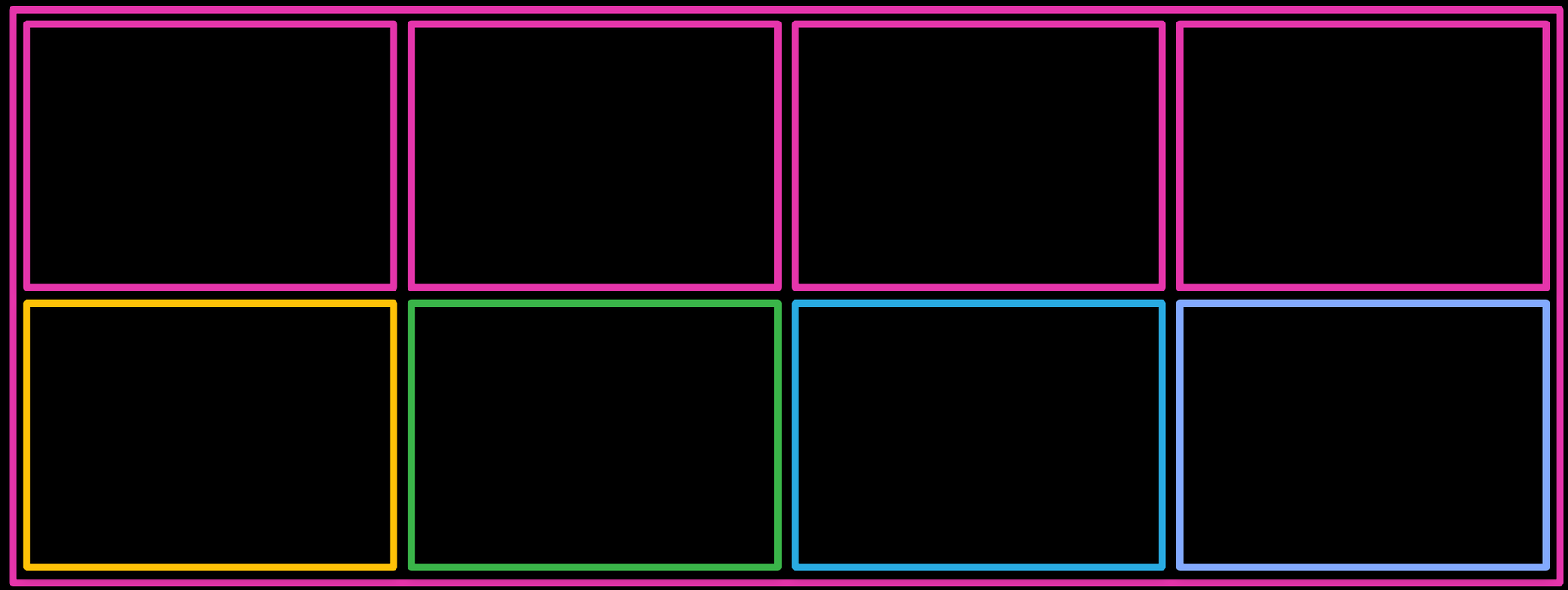
Application Complexity



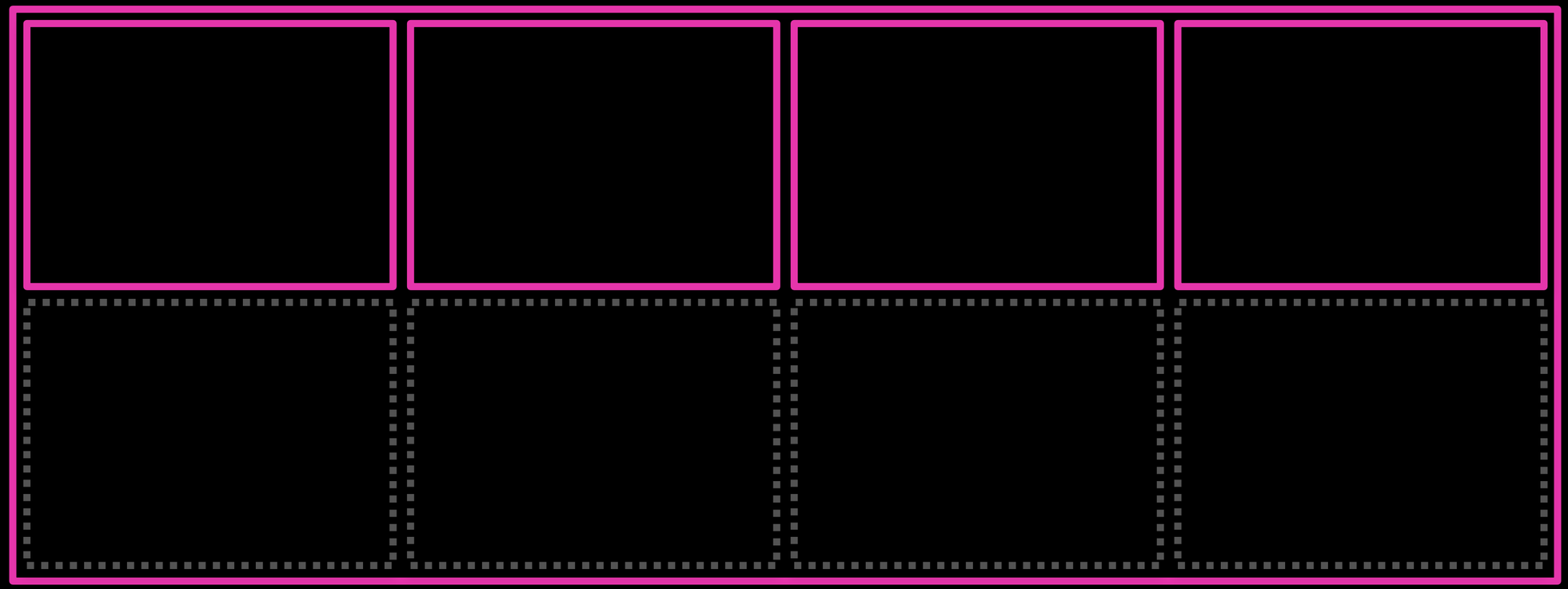
Organization Complexity

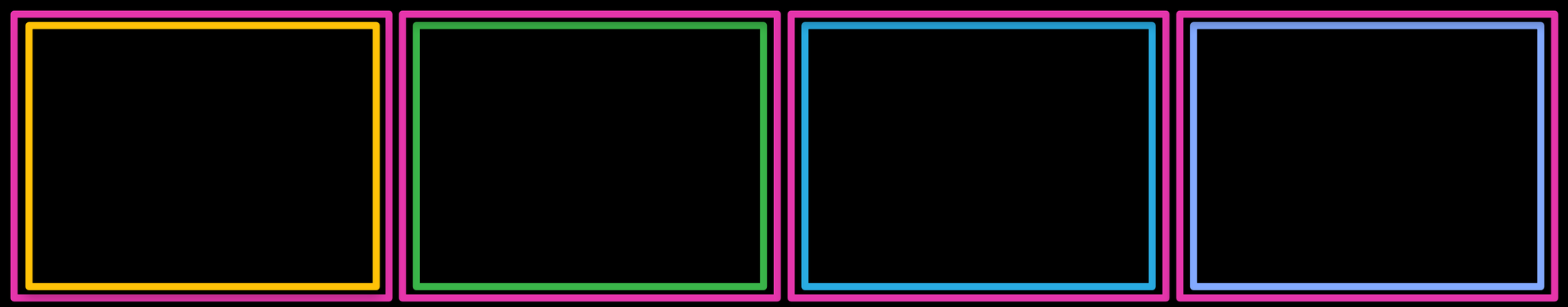
I thought this deployment would have went better

Module Federation

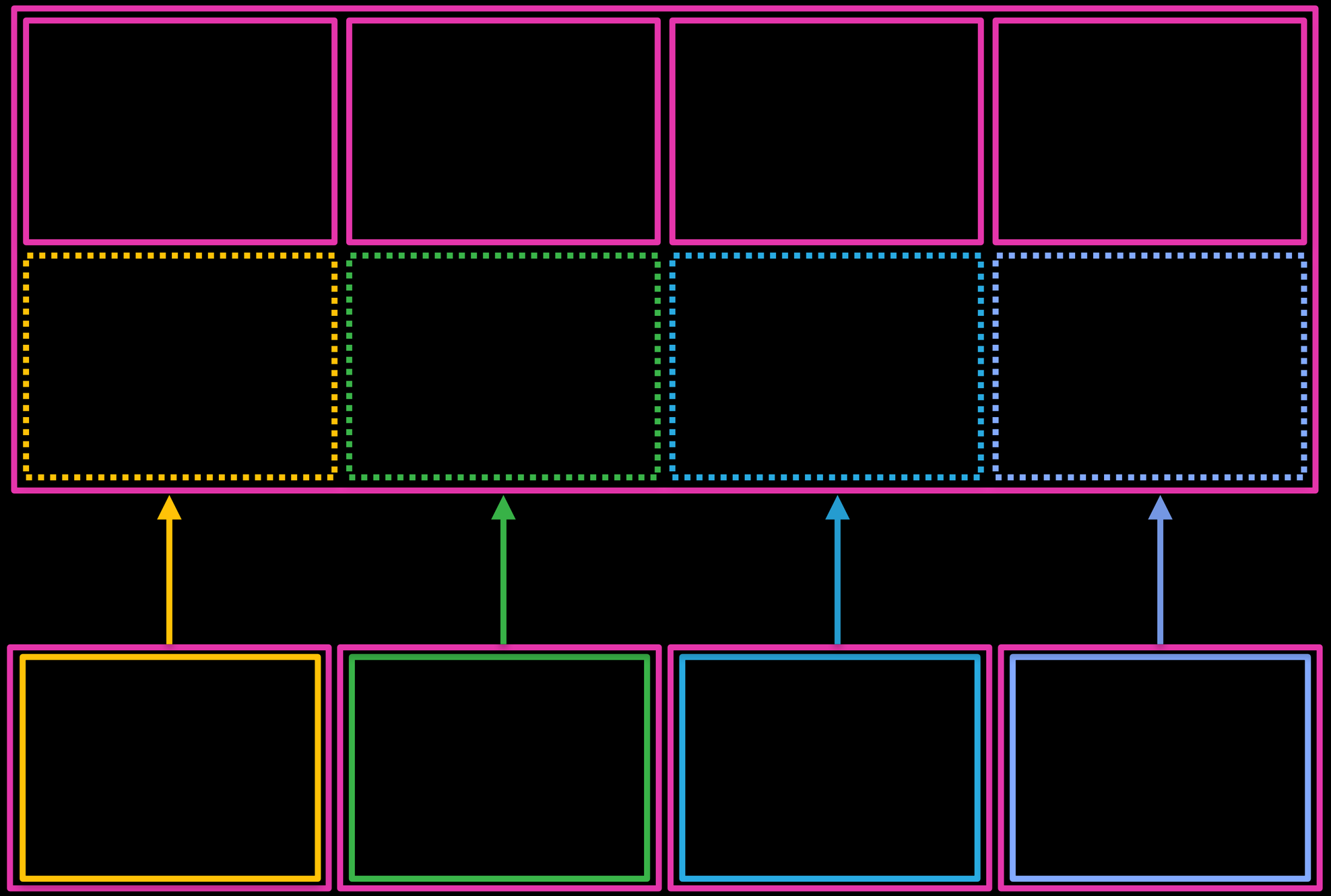


Organization Complexity



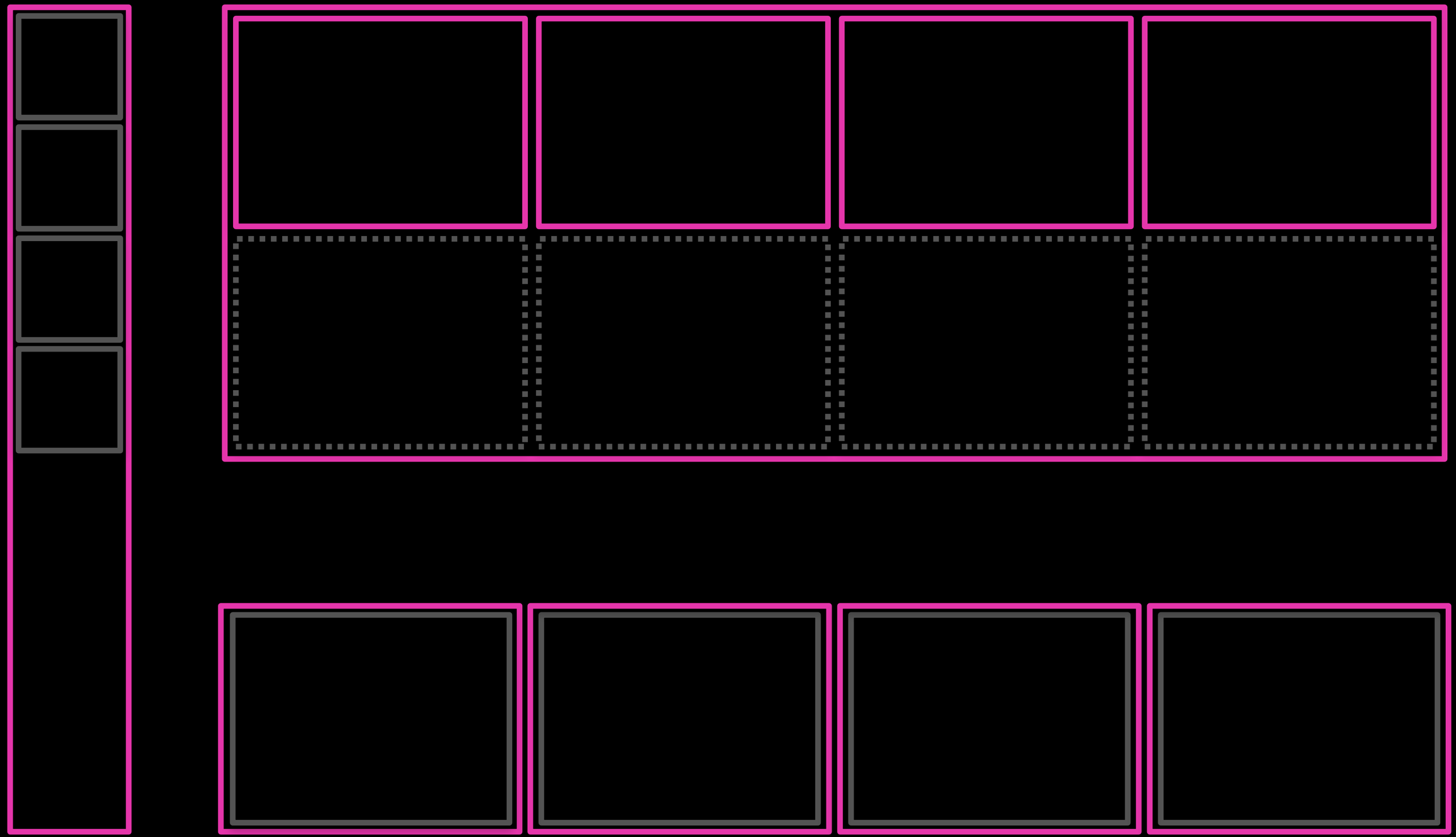


Extraction

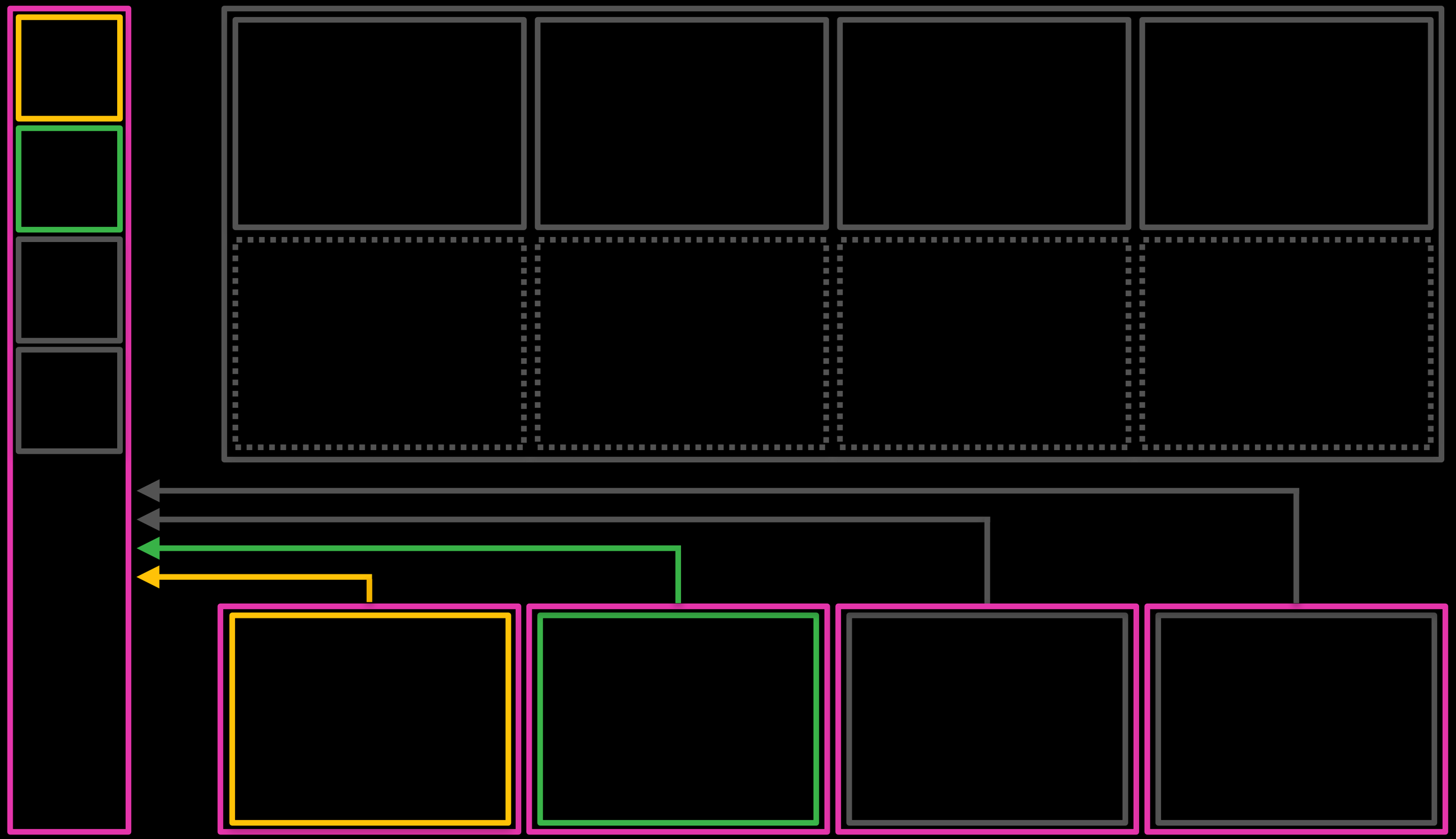


Consumption

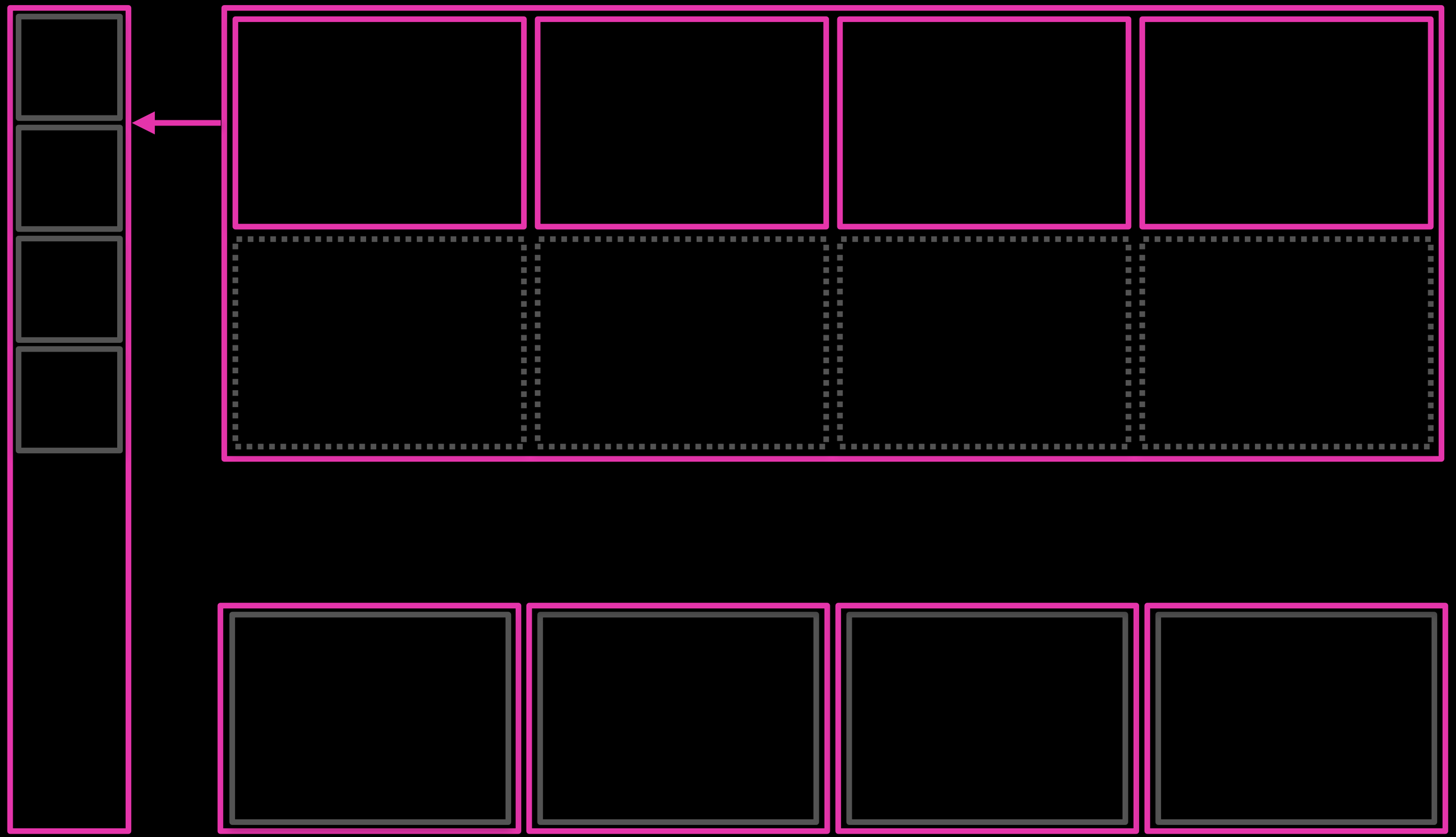
Module Discovery



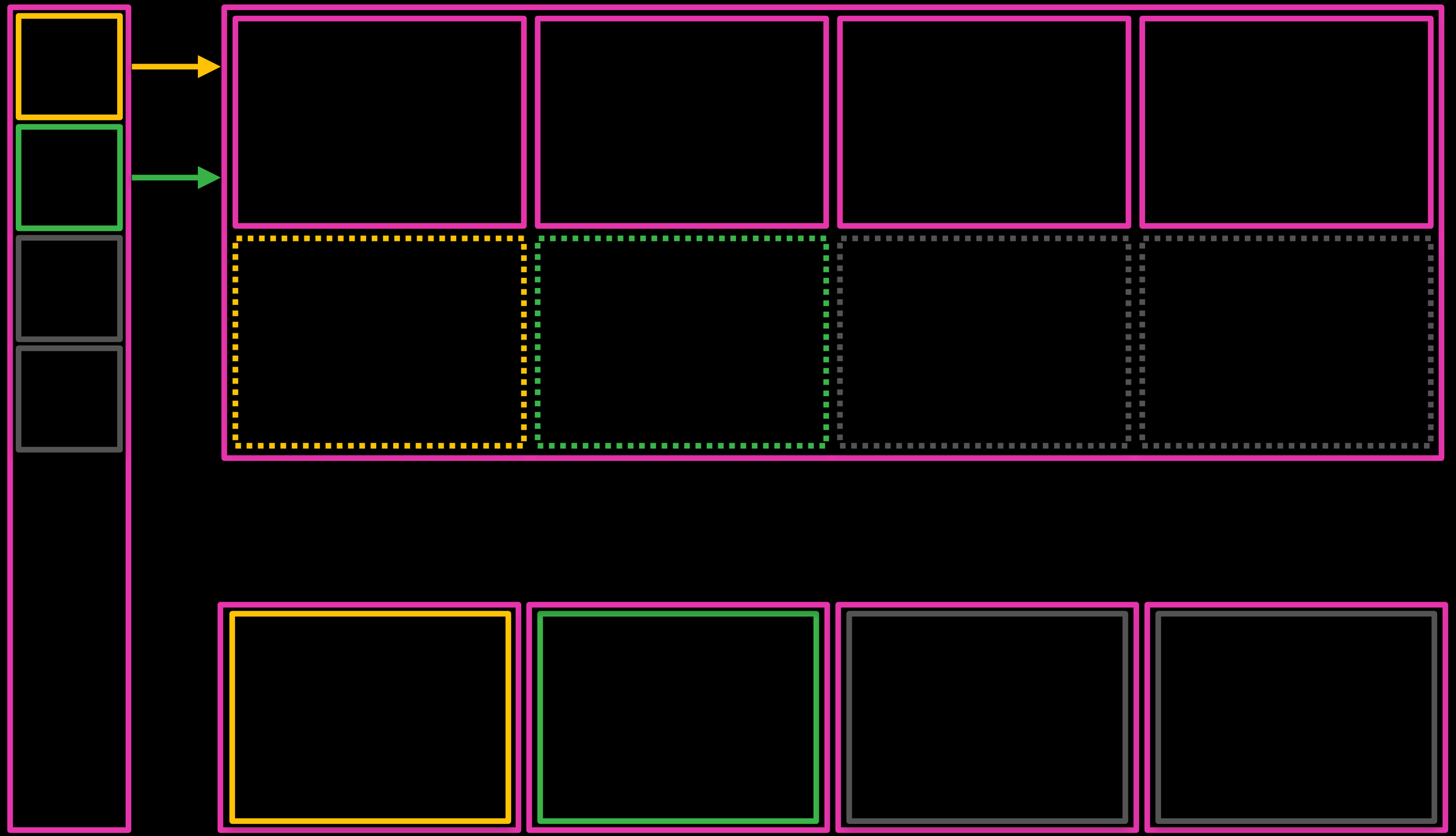
The Ledger™



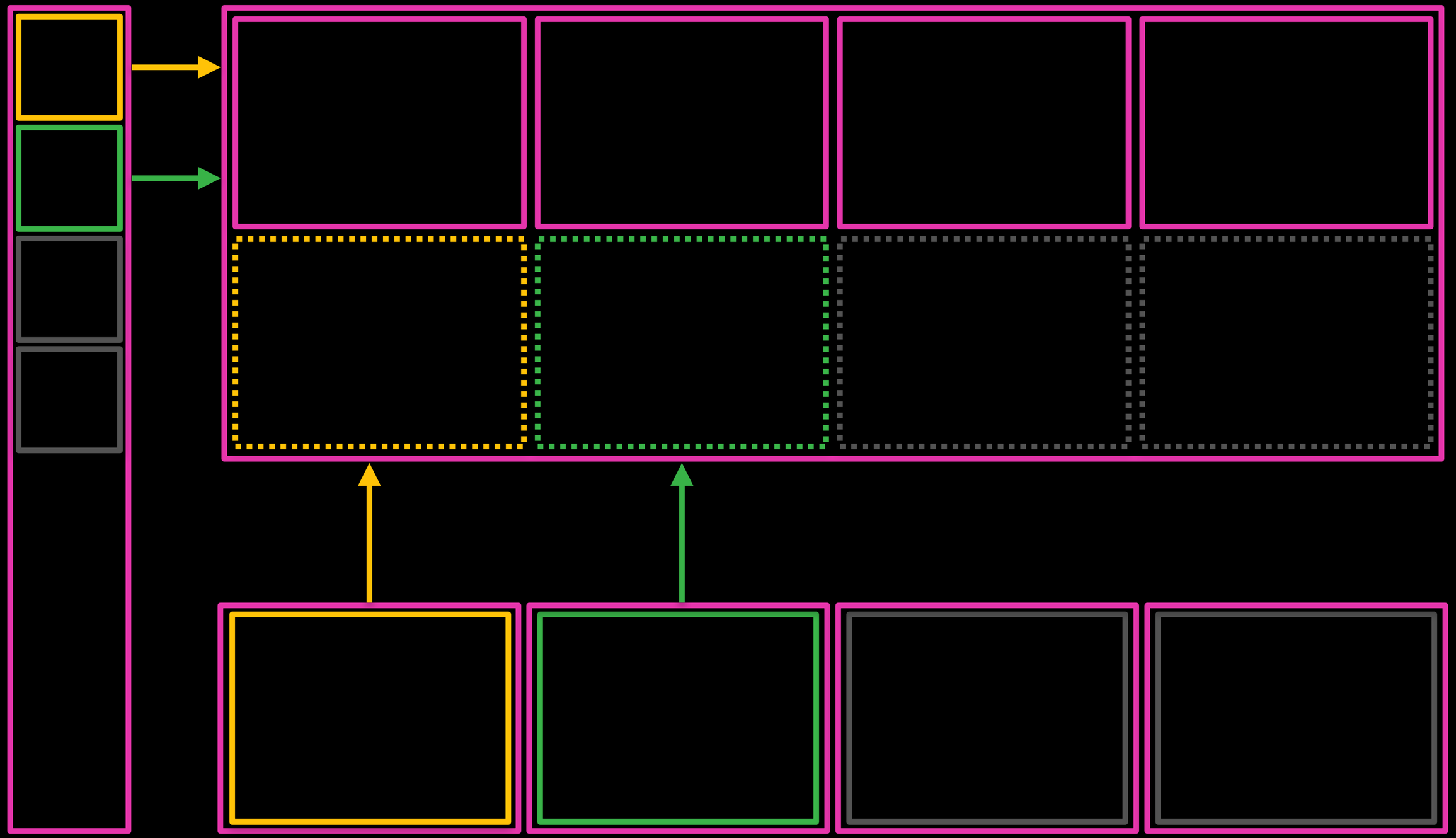
Module Registration



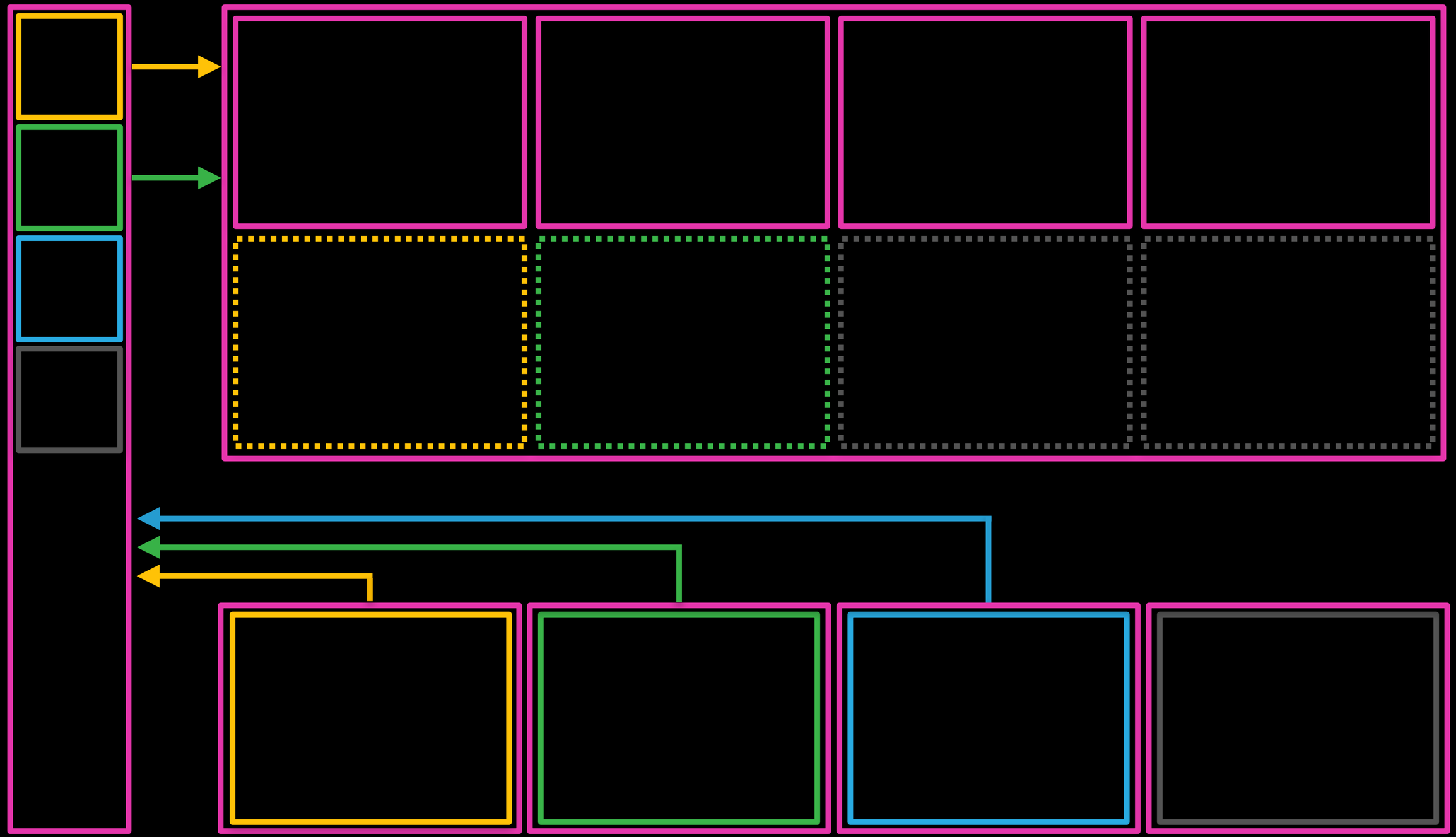
Discovery Query



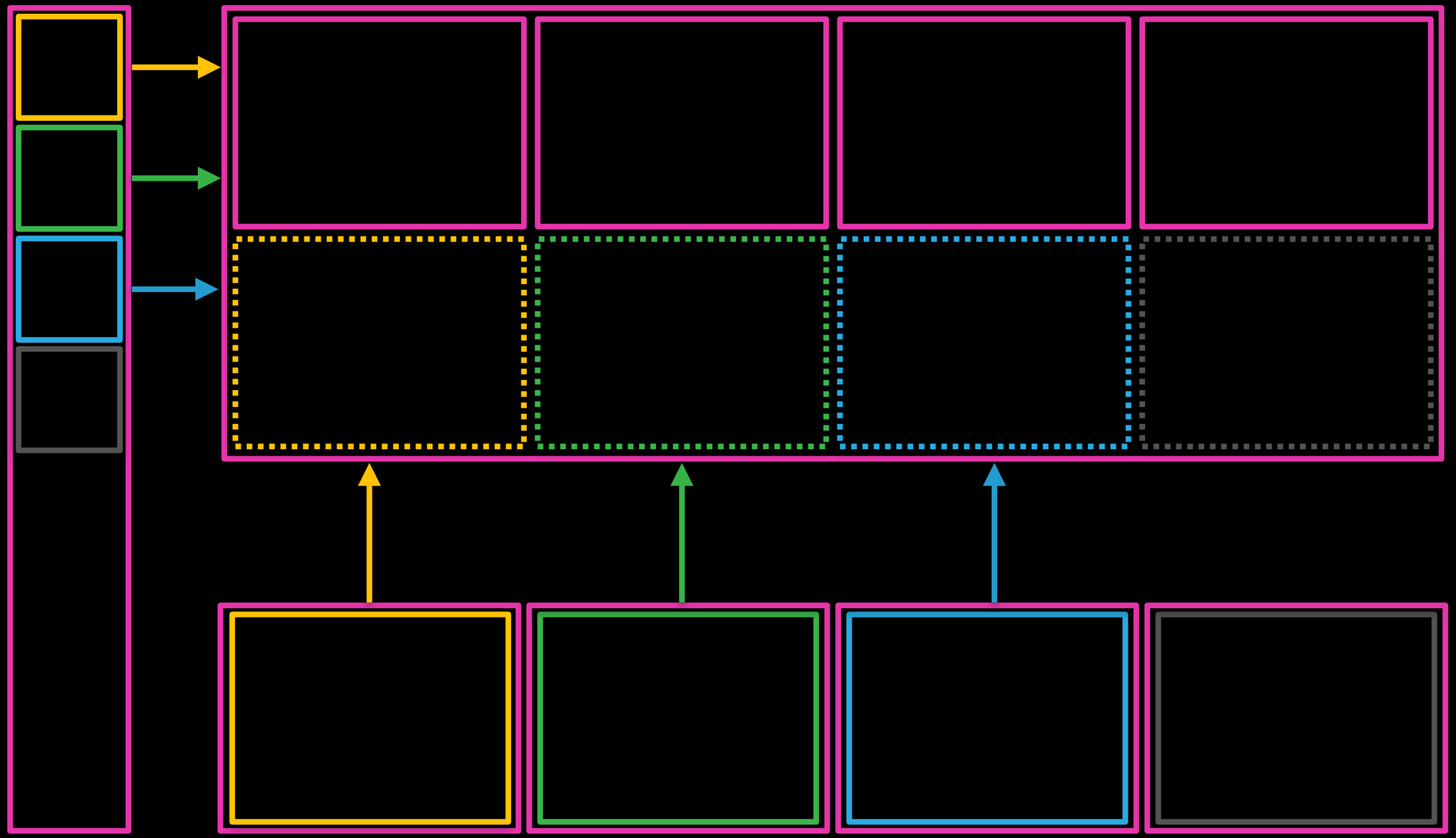
Module Discovery



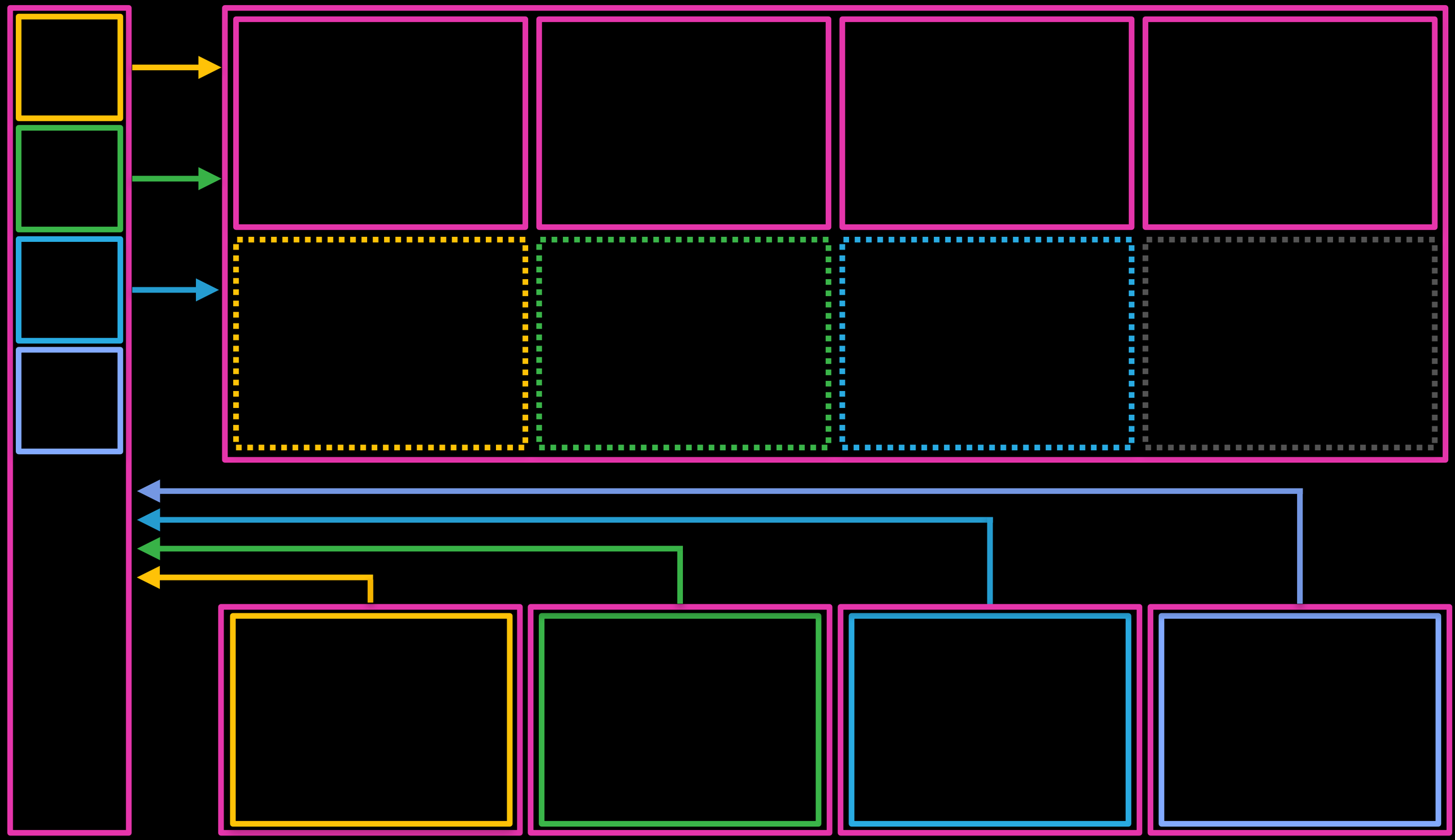
Module Consumption



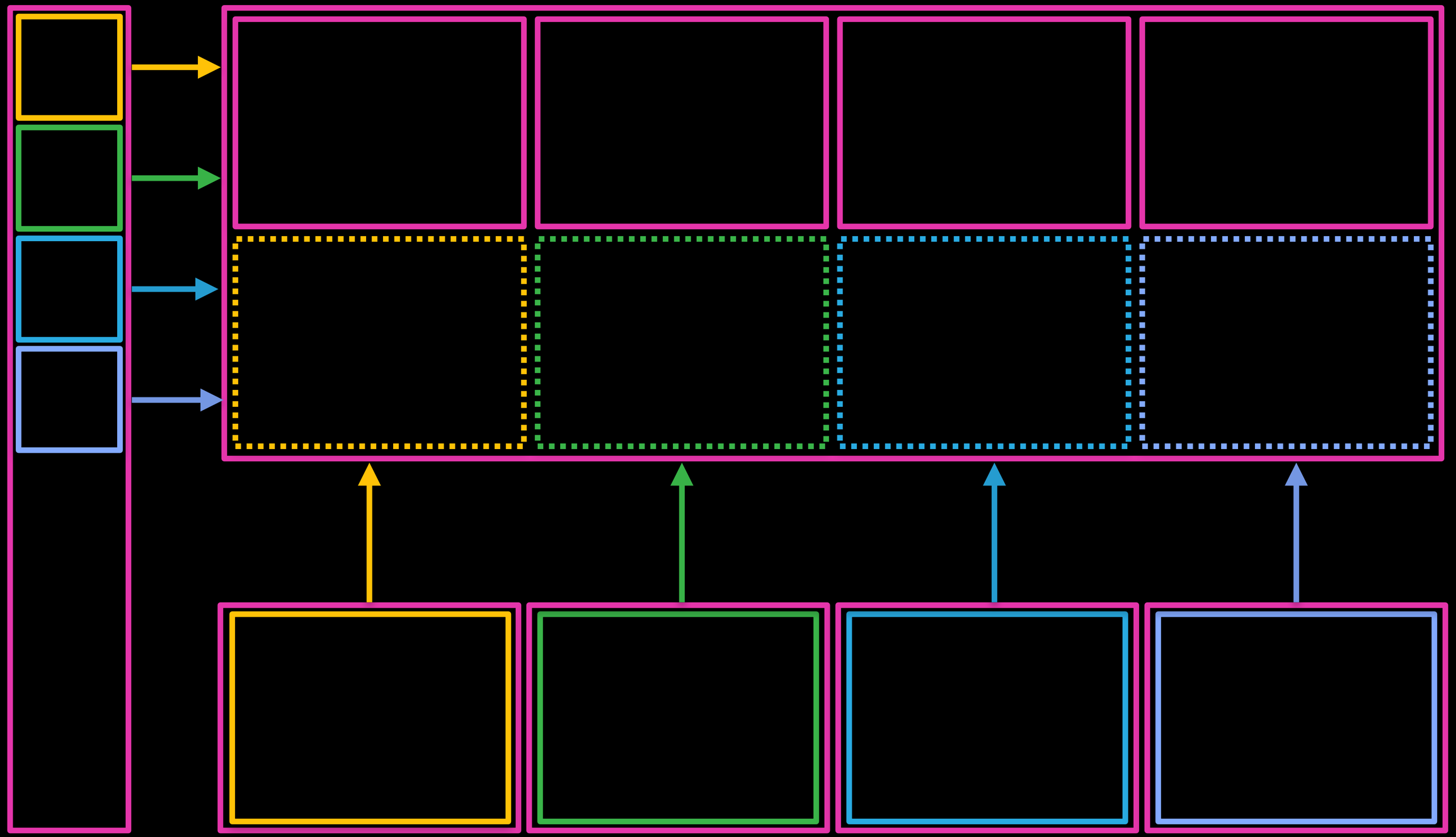
New Module Registration



New Module Consumption

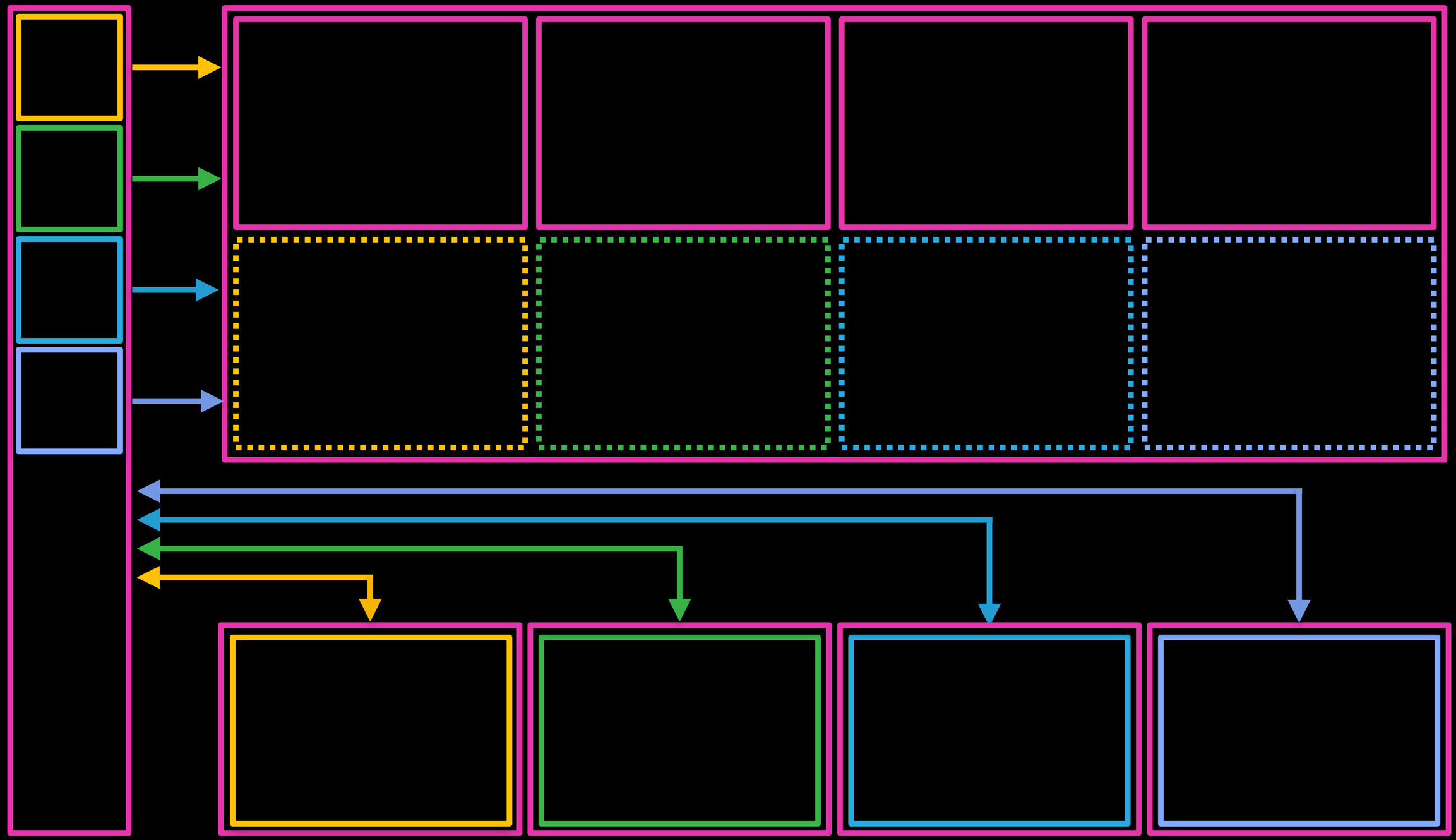


Another Module Registration

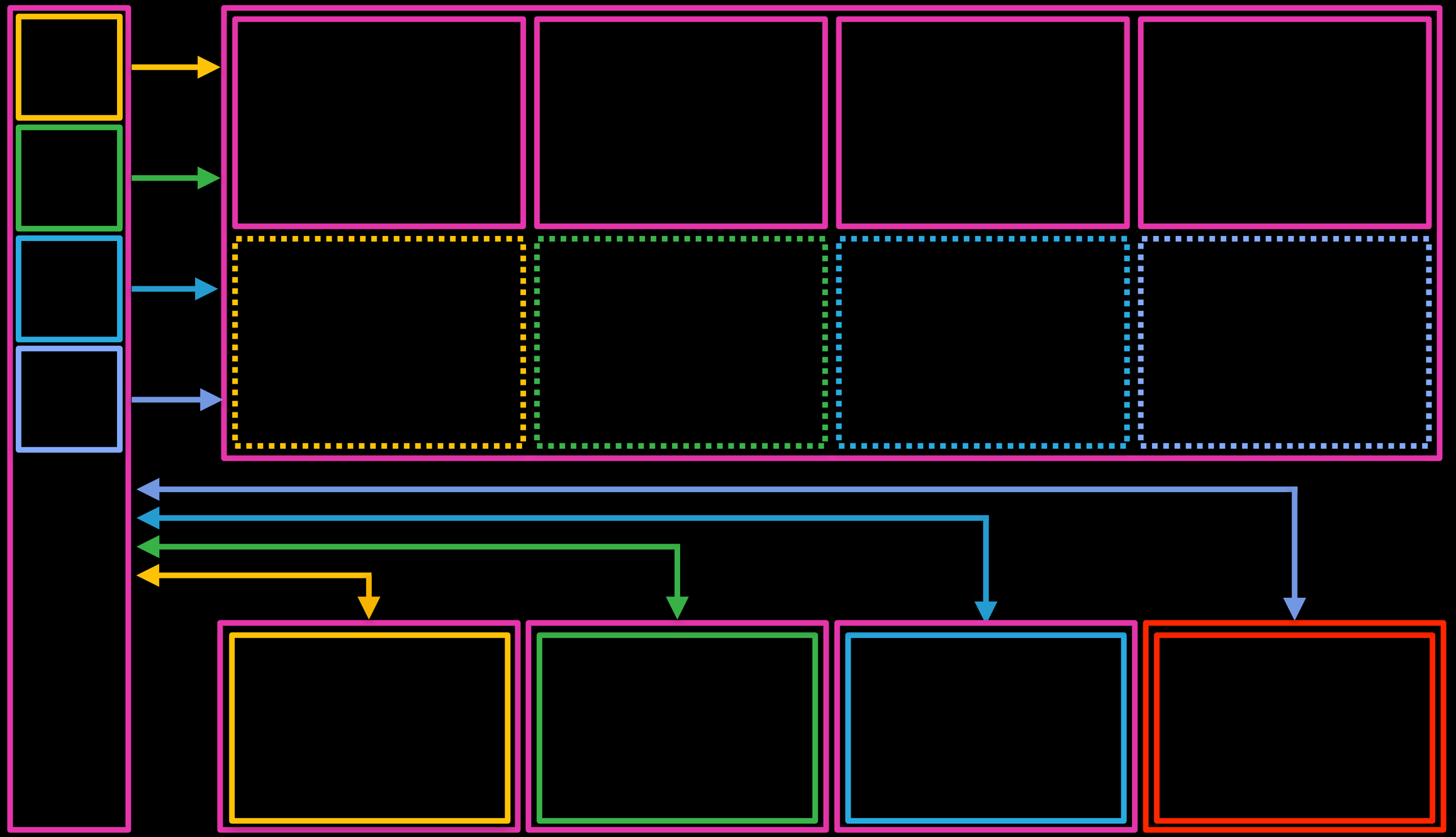


Another Module Consumption

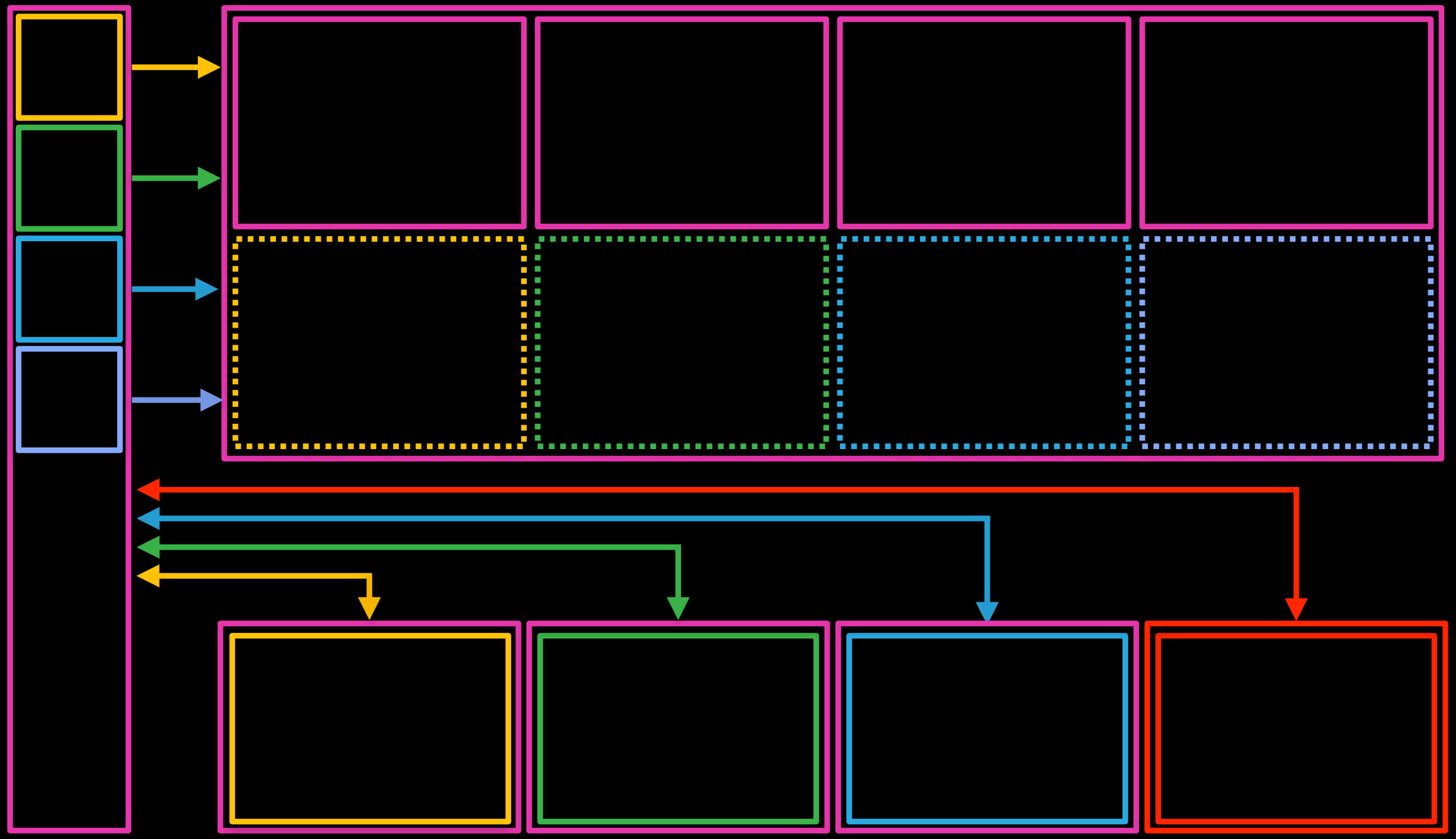
Health Check



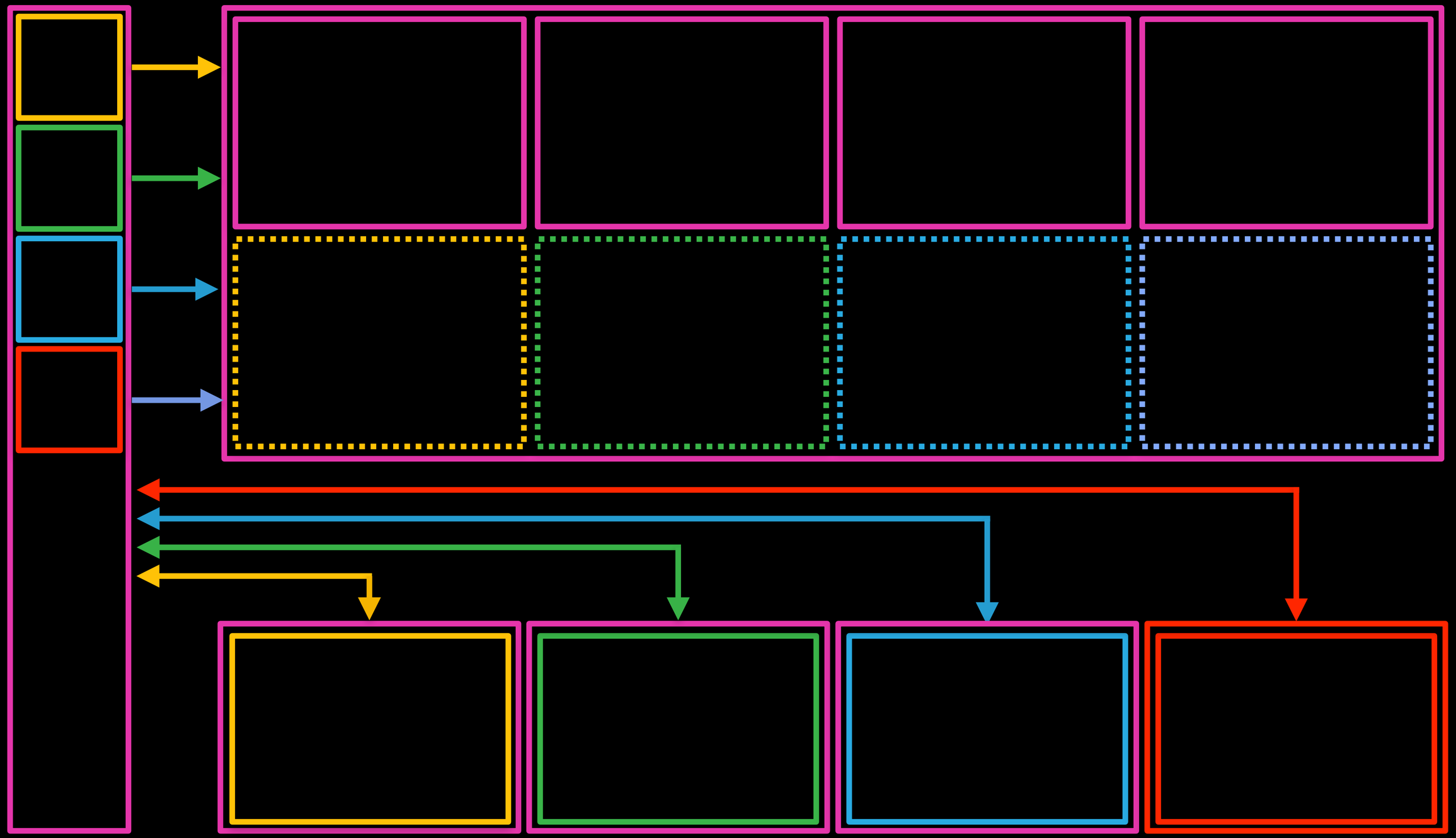
Systems Stable



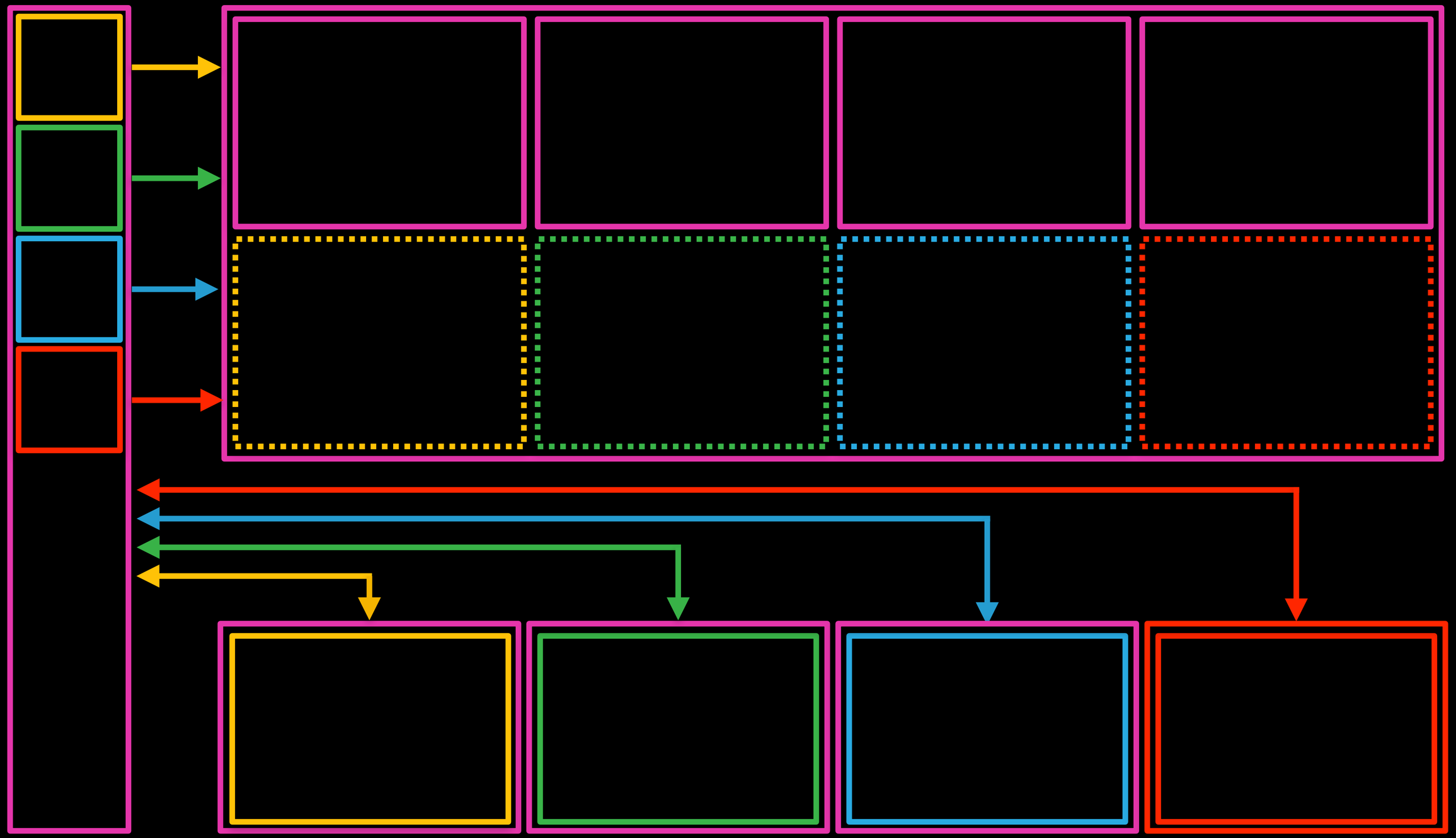
Module Offline



Offline Detection

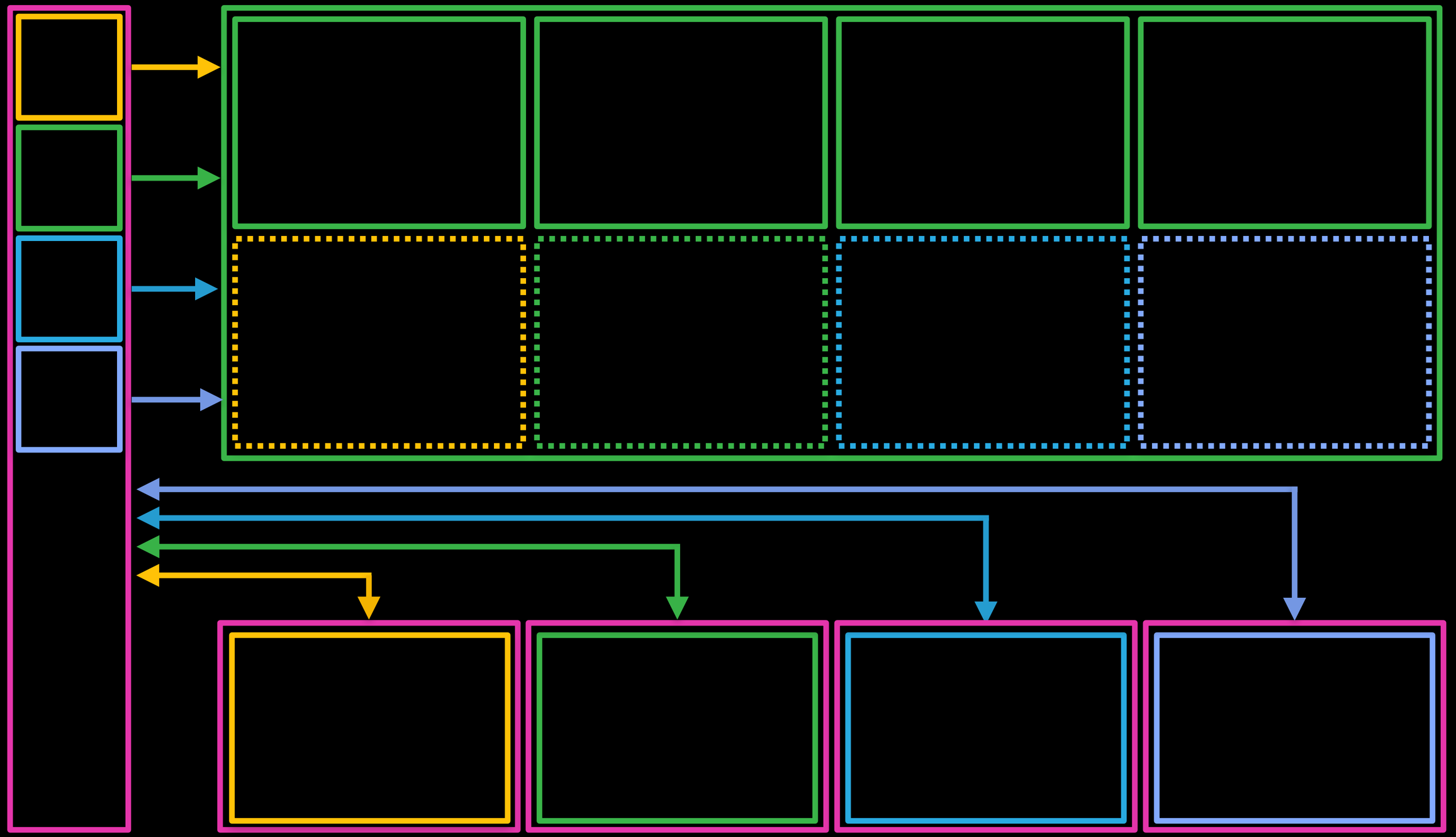


Ledger™ Updated

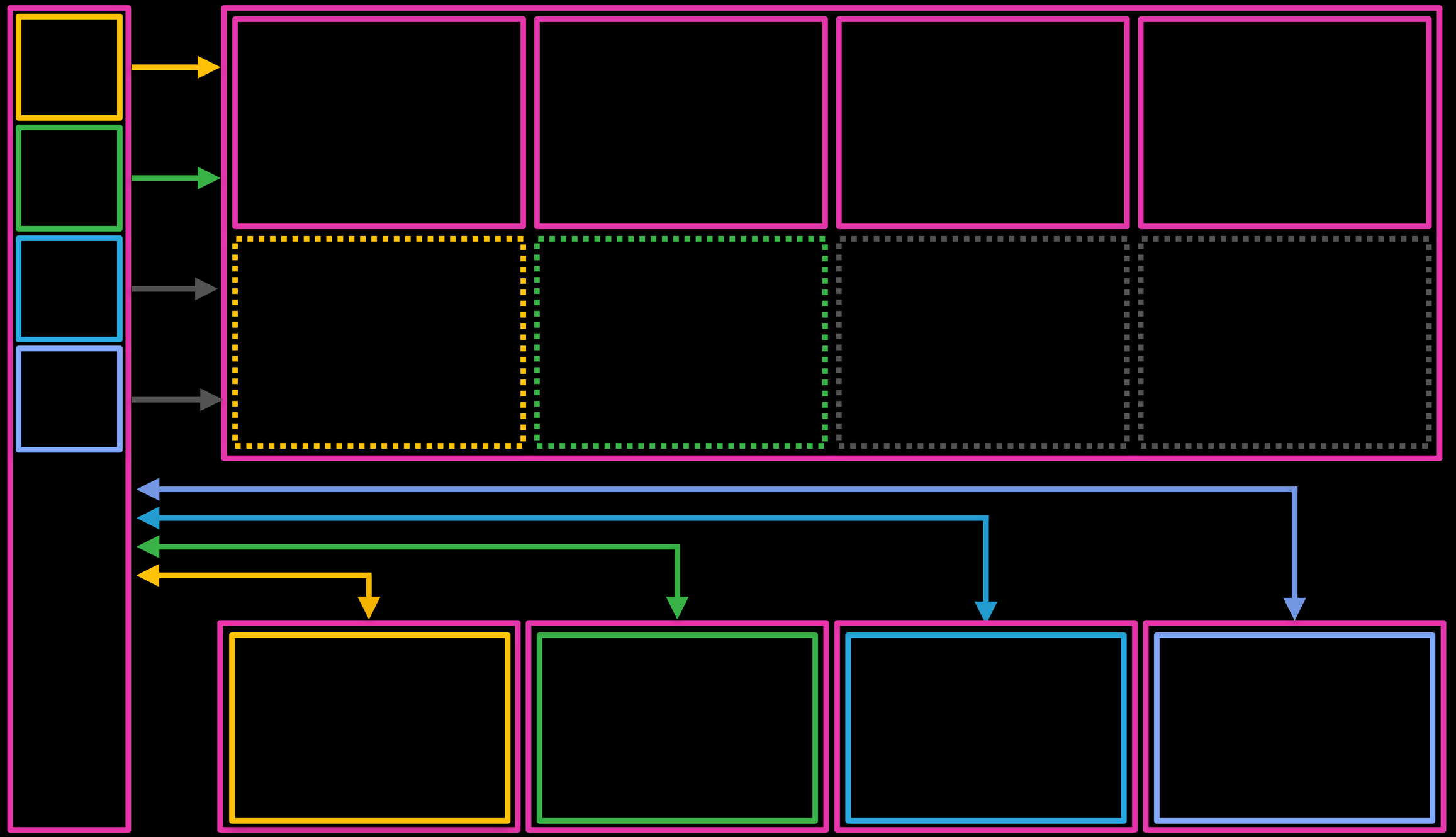


Consumer Notified

Authorized Consumption

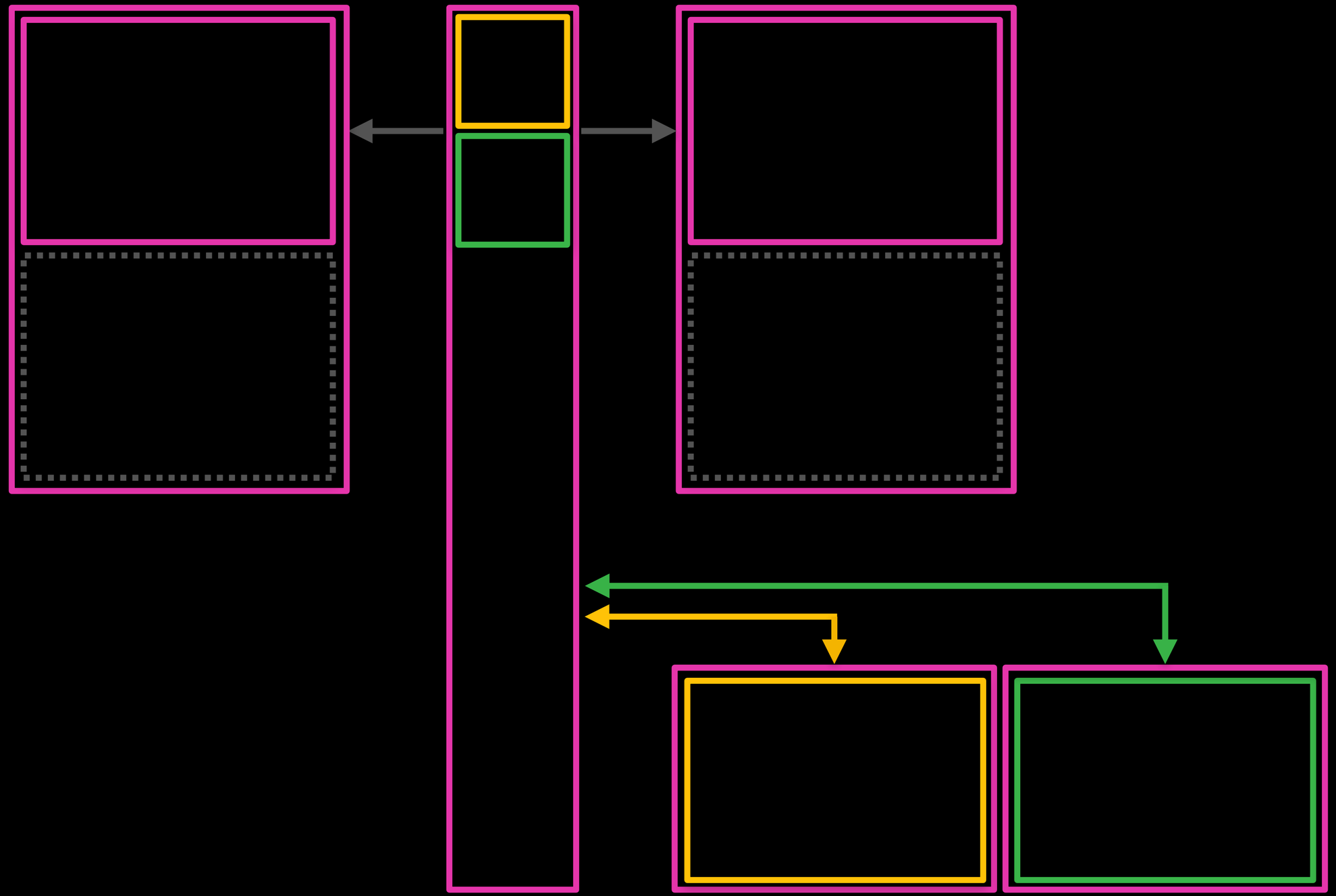


Premium Con$umer

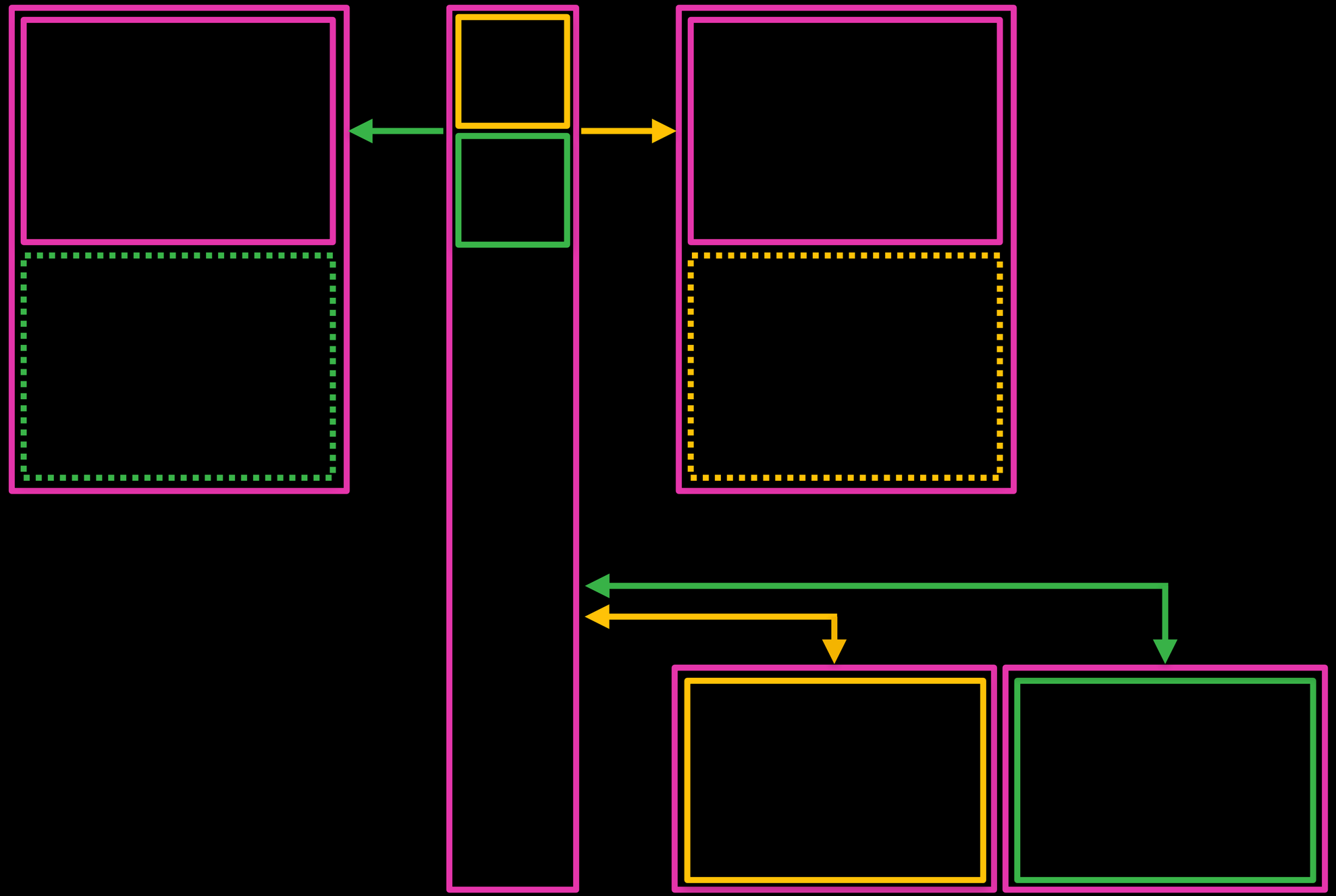


Restricted Consumer

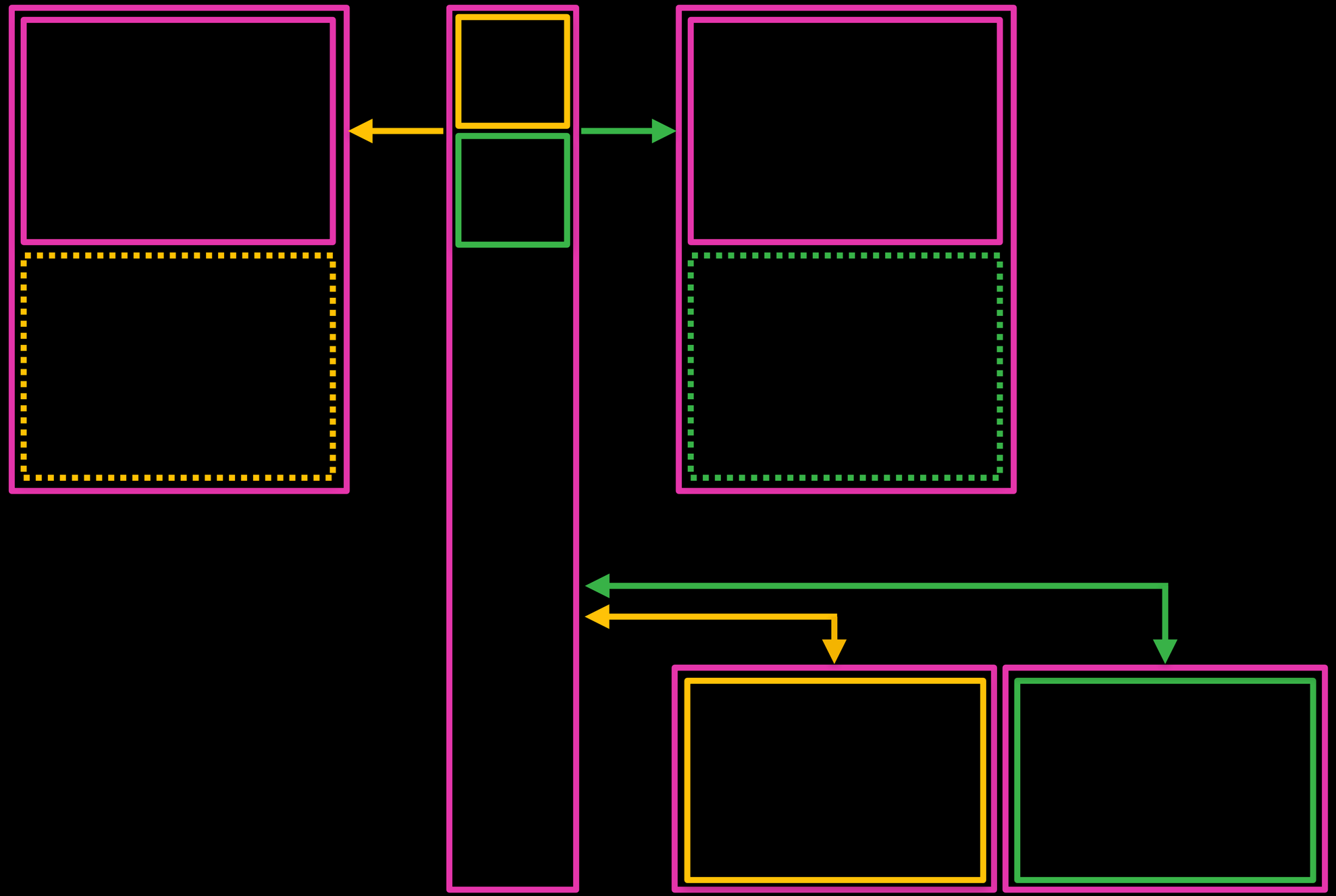
Split Testing



Two Variations and N Consumers



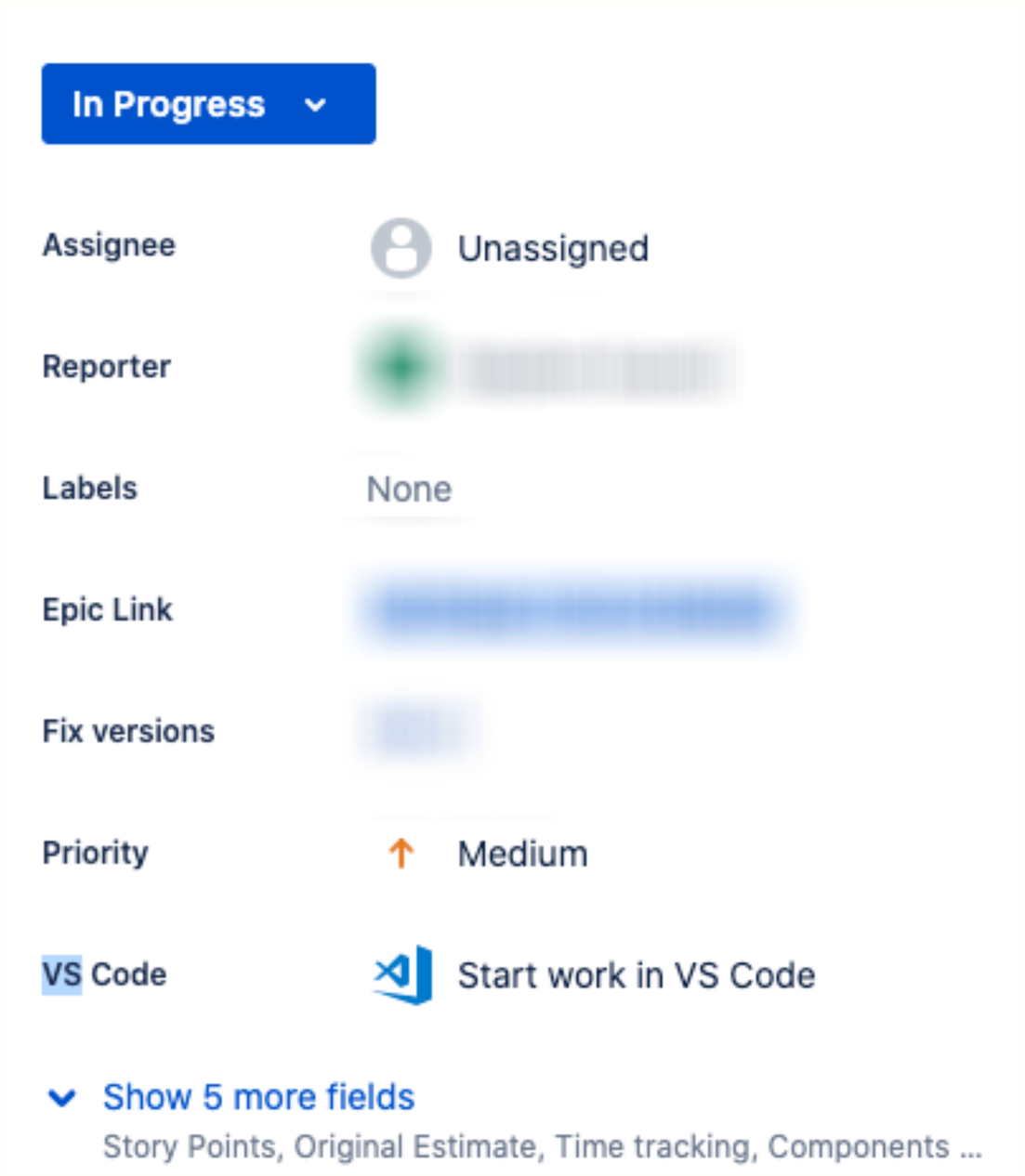
Modules Split Between Consumers



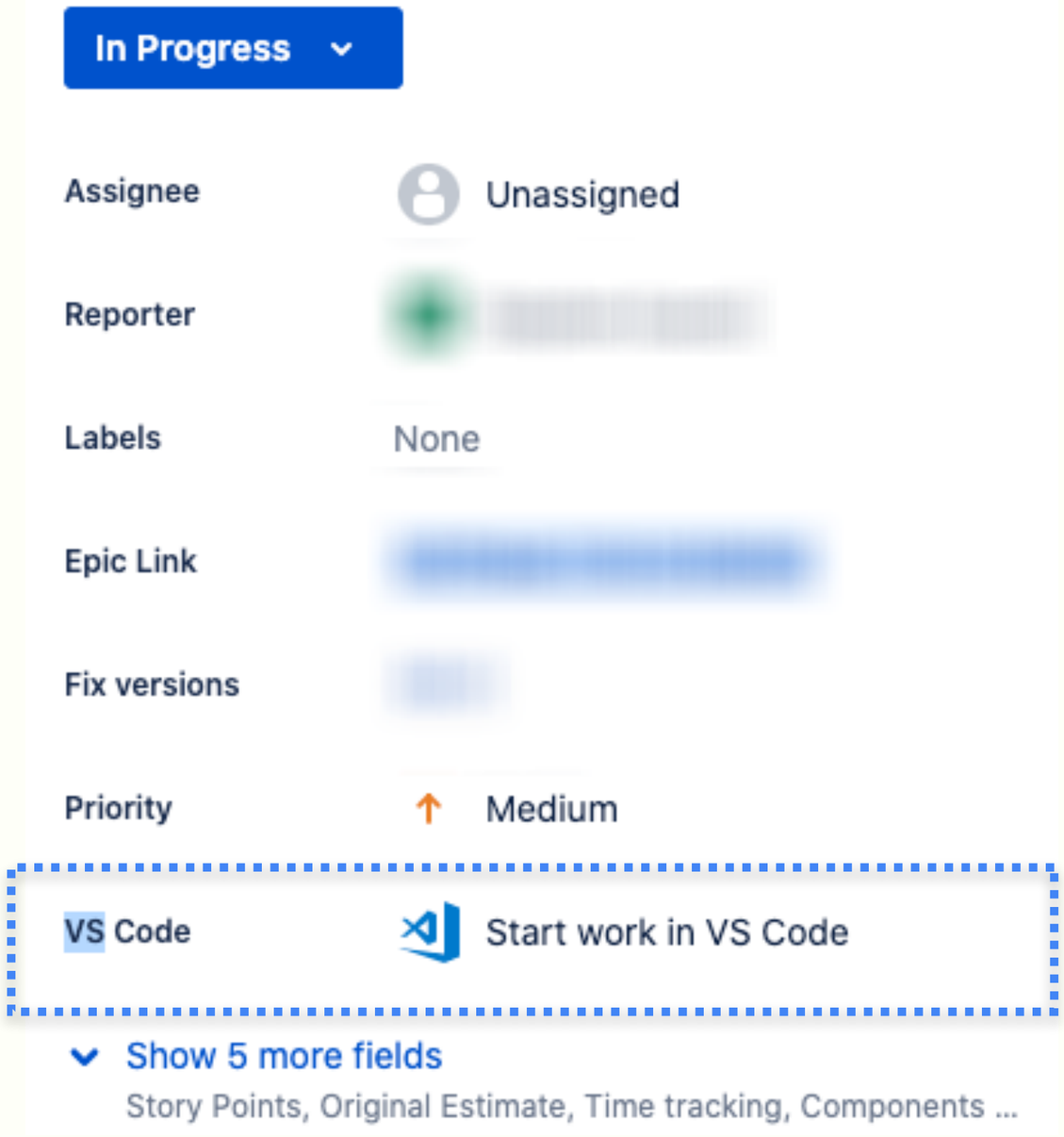
Modules Split Between Consumers

Module

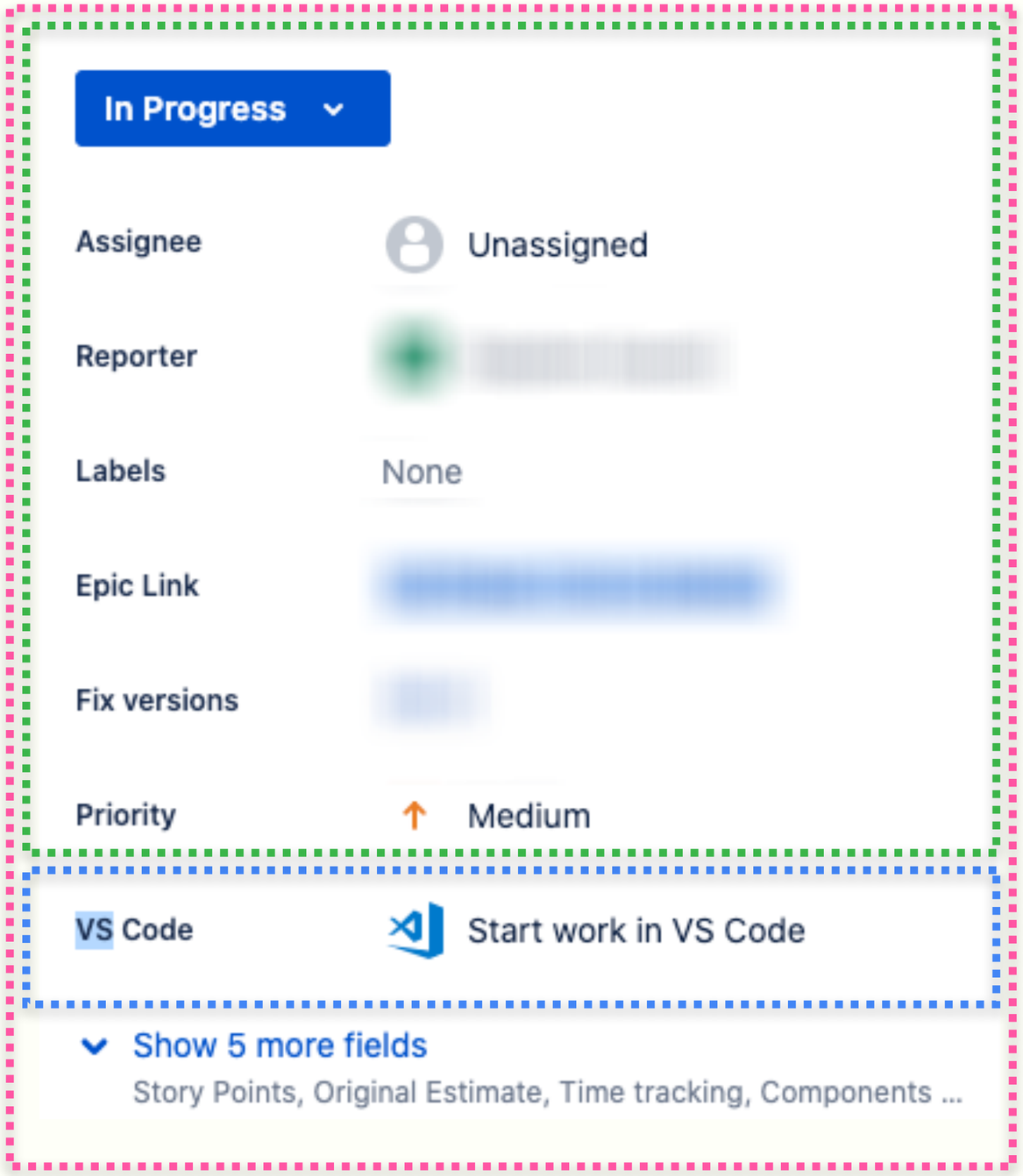
Inception


In Progress
Assignee
Unassigned
Reporter
Labels
None
Epic Link
Fix versions
Priority
Medium
VS Code
Start work in VS Code
Show 5 more fields
Story Points, Original Estimate, Time tracking, Components ...

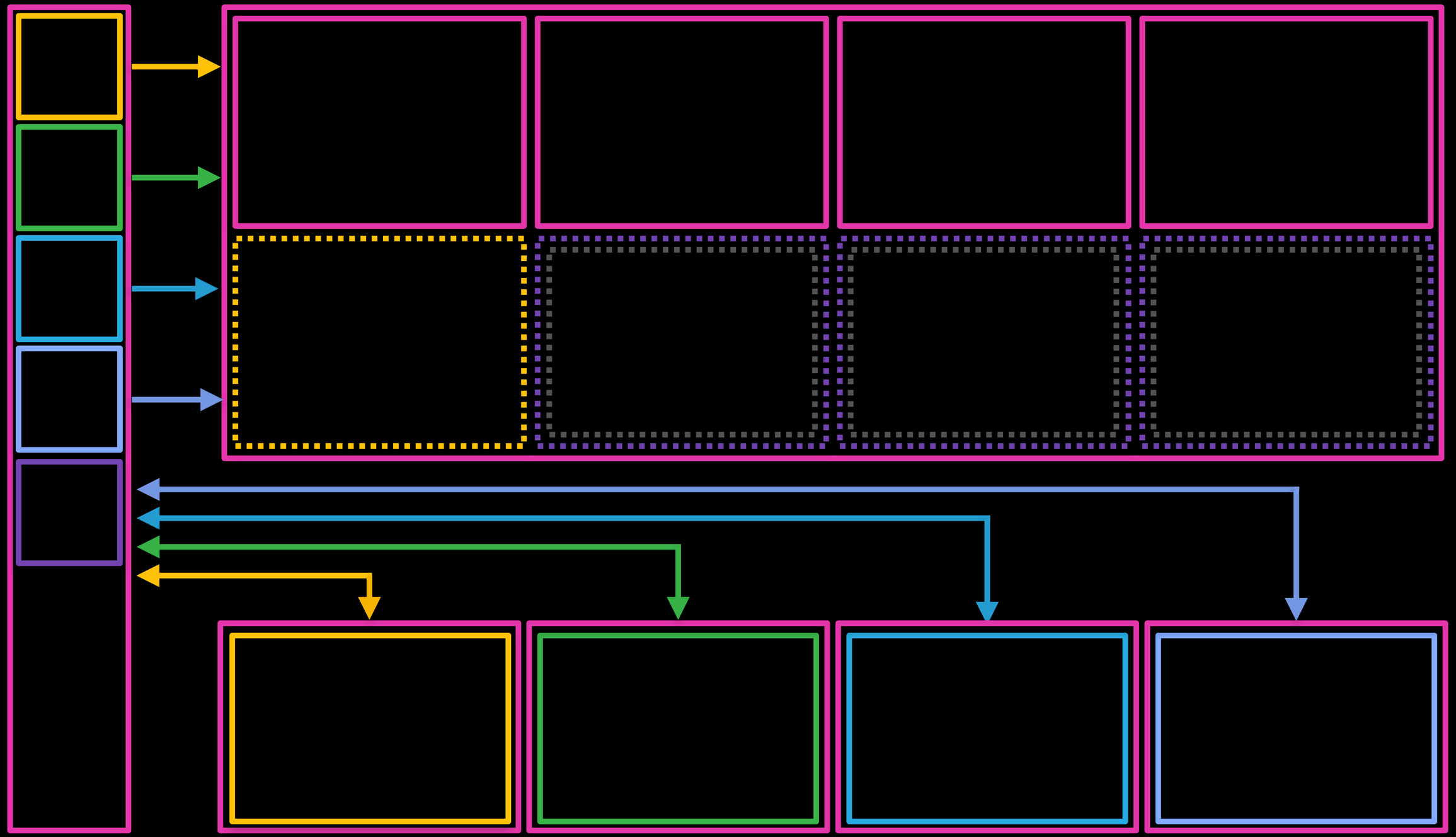
Meanwhile...



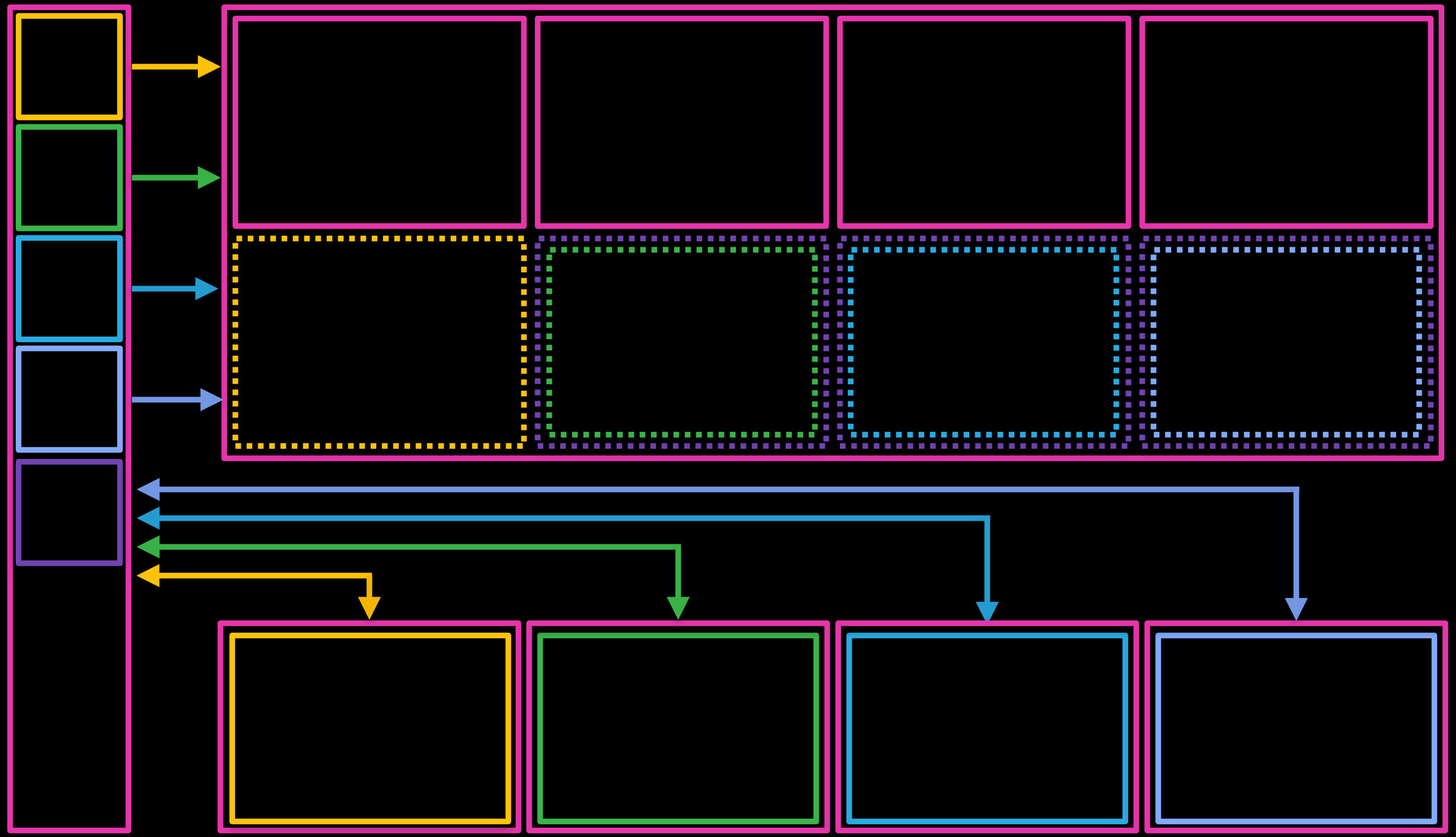
Hardcoded and Unwanted Feature



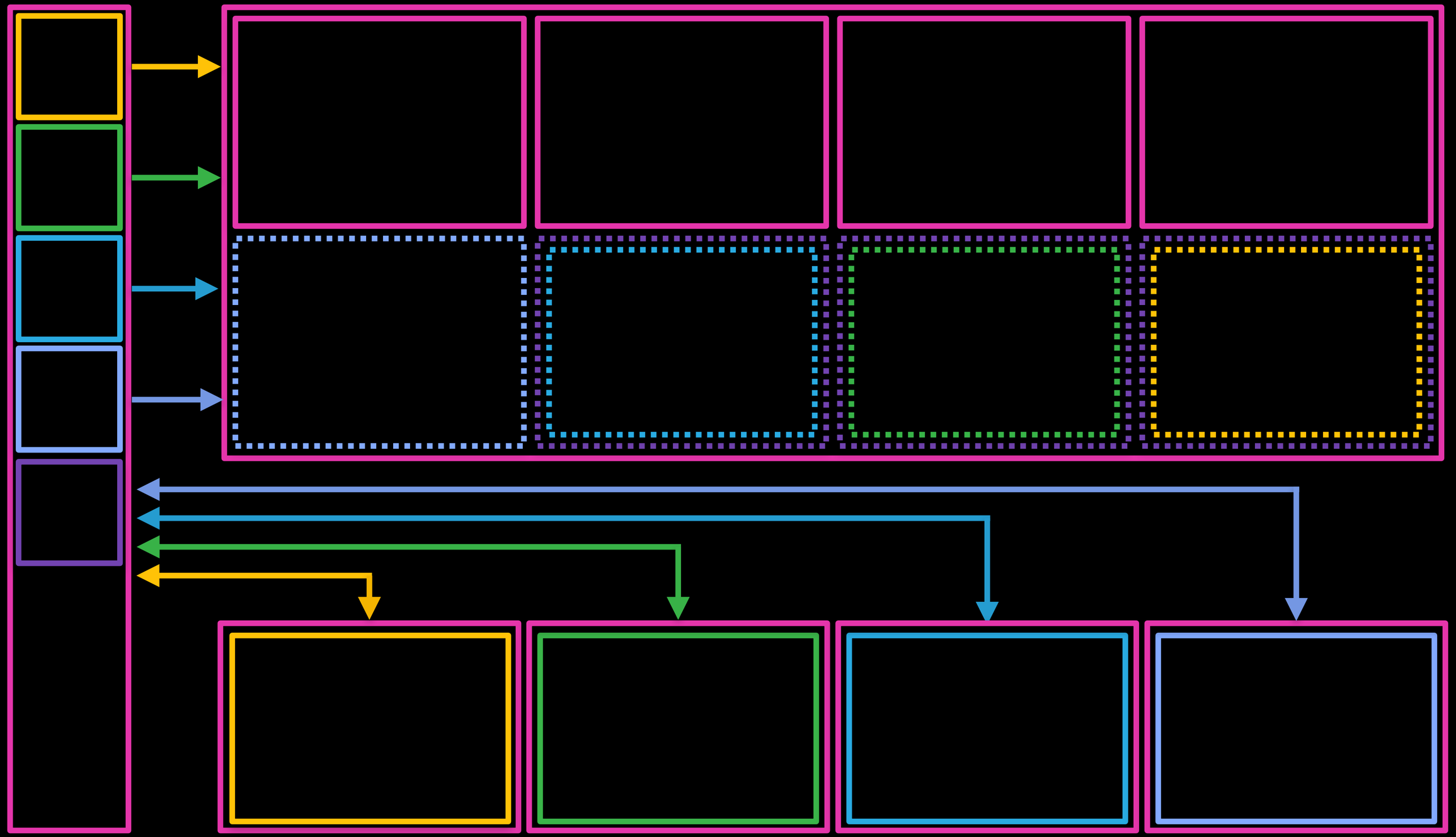
Dynamic Feature Consumption



Dynamic Module Outlet



Dynamic Module Consumption



Dynamic Module Consumption Variant