Extending the Composable Architecture Framework by Means of Normalized Systems Theory

Geert Haerens
Antwerp Management School, Belgium
Engie nv, Belgium
Email: geert.haerens@engie.com

Herwig Mannaert
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerpen.be

***Abstract*—In a fast-evolving world, companies require IT solutions that allow them to adapt swiftly to changing conditions. In 2020, Gartner introduced the Composable Architecture Framework as a guiding principle for creating application landscapes that are easily composable and recomposable, thereby supporting change. The Normalized Systems theory is about the creation of evolvable modular software. The concepts presented in the Composable Architecture Framework resonate with Normalized Systems. A closer analysis of the framework through Normalized Systems theory reveals that Gartner's framework lacks precision. As such, following the guidance of the framework will insufficiently protect companies from change, both outside and inside the organization. By extending the Composable Architecture Framework with Normalized Systems theory, a more evolvable framework emerges.**

Keywords*—*Normalized Systems Theory; Composable Architecture; Packaged Business Capabilities.

I. Introduction

This paper is an extended version of [1]. We elaborate on the concepts behind the Composable Architecture Framework (CAF) and Normalized Systems (NS) theory and extend our criticism of the original paper, providing guidance on how to improve CAF.

For many years now, a death wish toward the monolithic application has been declared. Monolithic applications are difficult to change and unsuitable in our fast-moving world. Many paradigms have been proposed over the years to split applications into smaller, modular parts. With the rise of the Internet and faster network speeds, the physical distribution of those parts has become a reality. The Distributed Computing Environment (DCE) [2] proposed modules encapsulating functionality that could be activated via Remote Procedure Calls (RPC). An essential benefit to splitting applications into smaller, independent callable modules is re-use. This resonates with McIlroy's dream, expressed during the 1968 NATO conference on Software Engineering, where *" ... I expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, [the user] should be able to regard components as black boxes safely."*.

SAP ERP (Enterprise Resource Planning), for instance, uses this approach to allow calling SAP functions from other systems via Remote Function Calls (RFCs). SAP re-baptized RFC to BAPI, Business Application Programming Interface, to focus even more on encapsulated functionality. In the past couple of years, we have seen a shift from encapsulated functionality toward technology, meaning that today, the talk of the town is about REST APIs, message queues, and event systems as a means of implementing integration between modules but without paying attention to the functionality.

In 2020, Gartner [3] introduced the notion of PBC to create Composable Applications. They proposed the CAF [3] as a guide for properly encapsulating functionality and using technologies that enable recomposition and evolution. Their approach connects the previous focus on functionality with today's emphasis on technology.

Normalized System theory (NS) [4], originating in software development, put forward the necessary conditions for the evolvability of modular structures.

When analyzing the CAF with NS, one starts noticing that CAF lacks precision. Following the guidance outlined by CAF will not be sufficient to protect a company's IT systems from change. This brings us to the problem statement we aim to address: How can NS help extend CAF to address the identified weaknesses?

The paper is structured as follows: In Section II, we will introduce Gartner's Reference Architecture for Composable Business Technology, followed by Section III, that introduces NS. In Section IV, we refer to related work, and in Section V, we analyze and criticize the framework through NS. In Section VI we have our findings validated by a focus group and in Section VII we will propose extensions to CAF to improve the overall robustness against change. The paper is wrapped up in Section VIII.

II. Gartner's Reference Architecture for Composable Business Technology

In 2020, the world was struck by COVID. In addition to human loss and suffering, businesses were severely disrupted by this crisis. Gartner noticed that companies with a modular application approach could adjust swiftly to external conditions by quickly and safely assembling, disassembling and reassembling applications as the world required. In November 2020, Gartner published their Reference Architecture for Composable Business Technology [3], that investigates the conditions required for a platform to facilitate the composition and re-composition of applications. The necessary ingredients for such a platform are PBCs as the essential modules, an application composition experience that allows custom assembly

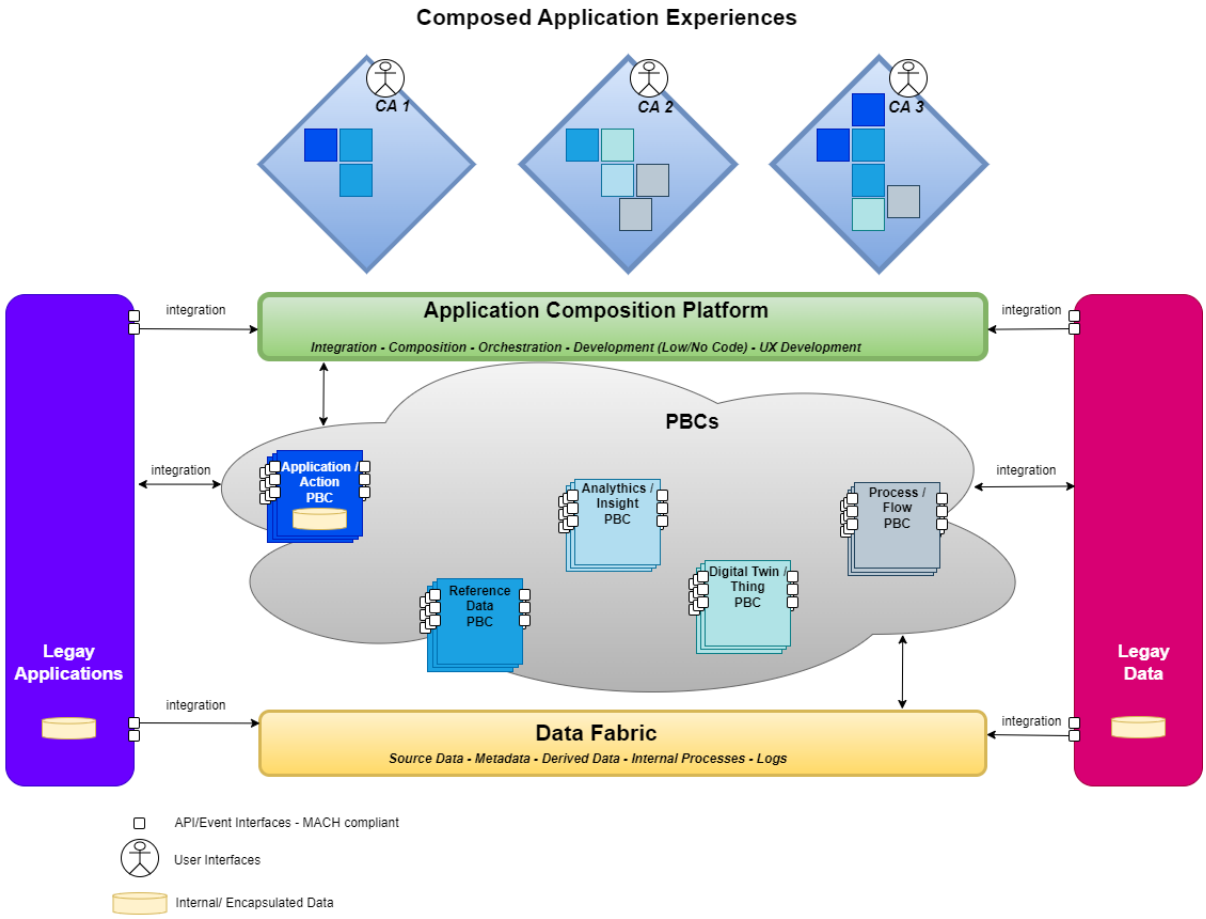

Fig. 1. Overview of the CAF [5].

of PBCs via low-code and no-code, an application composition platform that enables development and deployment of the newly composed applications, and a data fabric that provides easy access to data and analytics.

The PBCs are modules that encapsulate a well-defined business capability (BC) - recognized by the intended business users - and must adhere to the following conditions:

- They are modular and cohesive.
- They are autonomous (can run independently) and have minimal dependencies with external components.
- They allow orchestration as they can realize a process flow across PBCs (via APIs, events, etc.).
- They are discoverable and easily accessible and recognizable by those who require them.

The above definition of a PBC is heavily focused on the ”P” — how it should be packaged — and less on the ”BC” — the business capability. In Section V, we will see that the defining BCs is not straightforward and that the lack of definitions and guidance from Gartner is a weakness of the PBC concept.

Defining the right level of business capability granularity is challenging (too large = monolith, too small = more complex to identify).

Gartner provides further guidance on the PBC definitions by defining PBC types.

There is the Application Type PBC that encapsulates both data and functions related to a well-defined business capability. Application Type PBC can be used to create fully expressed (=autonomous and encapsulating a full context) PBCs or basis business function PBC (= not autonomous and encapsulating a part of a context). Application Type PBC can create pseudo-PBCs that act as APIs to existing monolithic applications. They encapsulate the legacy application, allowing them to participate in PBC composition, but they lack some of the flexibility in true PBCs regarding evolvability.

Reference-type PBCs allow encapsulated access to data in the data fabric. They can access master data, metadata, or any data instance that represents a business object or a physical data container.

The Insight Type PBC allows the encapsulation of data analytics processes. It can perform analytic operations or apply AI models (ML, Deep learning, etc.) to data in the data fabric.

The Thing Type PBC encapsulates physical-world entities. It can be used to access and manipulate data from the real world (IoT).

There is the Flow Type PBC, that combines different PBC

in a specific order. Flow Type PBCs facilitate the creation of orchestrated PBCs that encapsulate a process.

The PBCs are activated via event channels and called via APIs. They can also provide different optional user interfaces (web, mobile, etc.).

In Table I, Gartner contrasts the organization of an application landscape in PBCs (combined with a composition and deployment platform) with traditional application landscapes. Figure 1 provides an overview of the CAF.

Gartner analyzed the main change drivers in their papers: adding new business capabilities and their associated PBCs. Gartner further emphasizes that a platform that conforms to the above specifications is insufficient. Business and IT must work closely together to define and implement the required business capabilities, requiring what Gartner calls fusion teams. These teams are essential in creating business-IT alignment and, thus, value creation.

To start with a composable architecture strategy, one must first know what PBCs are already in the company. They may already be present as fine-grained PBCs, or via applications that aggregate PBCs and are accessible via APIs. It is vital that investments in future technologies that can be used for PBCs can be used for that purpose. The individual PBCs must be accessible via APIs, not aggregated. For example, a SaaS solution integrating multiple PBCs should allow the specific use of platform PBCs without depending on the other PBCs.

New applications are composed of assembled PBCs, allowing faster delivery and updating. Updating a PBC in the catalogue should update all applications with that PBC as an active module.

In summary, the Gartner CAF *"presents the reference model for developing business applications that are modular, composable, easily adapted and ready for change."* [3].

III. Fundamentals of NS theory

Software should be able to evolve as business requirements change over time. In NS theory [6], the lack of Combinatorial Effects measures evolvability. When the impact of a change depends not only on the type of the change but also on the size of the system it affects, we talk about a Combinatorial Effect. The NS theory assumes that software grows indefinitely and undergoes unlimited changes over time. Under those conditions Combinatorial Effects harm software evolvability.

NS theory is built on the principles of classical system engineering and statistical entropy. In classic system engineering, a system is stable if it has bounded input and bounded output (BIBO). NS theory applies this idea to software design, as a limited change in functionality should cause a limited change in the software. In classic system engineering, stability is measured at infinity. NS theory considers infinitely large systems that undergo infinitely many changes. A system is stable for NS if it does not have Combinatorial Effects, meaning that the effect of change only depends on the kind of change and not on the system size.

In the context of composable architecture, NS theory provides a lens for evaluating whether PBC-based systems will remain stable as they evolve. As Gartner claims, the CAF should result in more evolvable and thus easier to change systems; compliance with NS seems necessary.

NS theory proposes four theorems and five extendable elements as the basis for creating evolvable software through the pattern expansion of these elements. The theorems are proven formally, giving a set of required conditions to follow strictly to avoid Combinatorial Effects. The NS theorems have been applied in NS elements. These elements offer a set of predefined higher-level structures, patterns, or "building blocks" that provide a clear blueprint for implementing the core functionalities of realistic information systems, following the four theorems.

The NS elements are the building blocks needed for any modular architecture, and thus also for the CAF.

A. NS Theorems

NS theory [6] is based on four theorems that dictate the necessary conditions for software to be free of Combinatorial Effects.

- Separation of Concerns
- Data Version Transparency
- Action Version Transparency
- Separation of States

Violating any of these four theorems will lead to Combinatorial Effects and, thus, non-evolvable software under change.

B. NS Elements

Consistently adhering to the four NS theorems is challenging for developers. First, following the NS theorems leads to a fine-grained software structure, that introduces some development overhead and may slow the development process. Second, the rules must be followed constantly and robotically, as a violation will introduce Combinatorial Effects. Humans are not well-suited for this kind of work. Third, the accidental introduction of Combinatorial Effects results in an exponential increase in rework.

Five expandable elements [7] [8] were proposed, that make the realization of NS applications more feasible. These elements are carefully engineered patterns that comply with the four NS theorems and that can be used as essential building blocks for various applications: data element, action element, workflow element, connector element, and trigger element.

- **Data Element**: the structured composition of software constructs to encapsulate a data construct into an isolated module (including get- and set methods, persistency, exhibiting version transparency, etc.).
- **Action Elements**: the structured composition of software constructs to encapsulate an action construct into an isolated module.
- **Workflow Element**: the structured composition of software constructs describing the sequence action elements should be performed to fulfil a flow into an isolated module.
- **Connector Element**: the structured composition of software constructs into an isolated module, allowing external

TABLE I
CONTRASTING TRADITIONAL WITH PBC APPLICATION LANDSCAPES (FROM [3])

Criteria	Traditional Applications	PBCs
Primary value	Business capability	Business capability
Primary access	User Interface (UI)	Programmatic Interface (API, event)
Scope	Many business objects	One business object
Internal architecture	monolith or modular	monolith or modular
Designed for	Business	Business and IT
Design priority	Stability	Agility
Delivered value	Business solutions	Recomposable business solutions
Production style	Project	Product
Essential tools	Customization included	Composition, added cost
Required IT skills	Customization, low	Composition, high
Cost	Bulk, some ”shelfware”	Componentized, tracks value
Governance	Sample	Complex
Internal data	”Owned”	”Owned”
Open for integration/composition	Partially, a secondary priority	Fully, primary design objective

systems to interact with the NS system without calling components statelessly.

- **Trigger Element**: the structured composition of software constructs into an isolated module that controls the system states and checks whether any action element should be triggered accordingly.

The element provides core functionalities (data, actions, etc.) and addresses the Cross-Cutting Concerns (CCCs) that each of these core functionalities requires to function correctly. CCCs cut through every element, requiring careful implementation to avoid introducing Combinatorial Effects.

C. Element Expansion

An application comprises data, action, workflow, connector, and trigger elements that define its requirements. The NS expander is a technology that generates code instances of high-level patterns for a specific application. The expanded code will provide generic functionalities specified in the application definition and will be a fine-grained modular structure that follows the NS theorems (see Figure 2).

The application’s business logic is now manually programmed inside the expanded modules at pre-defined locations. The result is an application that implements the required business logic and has a fine-grained, modular structure. As the code’s generated structure is NS-compliant, we know it is evolvable for all anticipated change drivers corresponding to the underlying NS elements. The only location where Combinatorial Effects can be introduced is in the customized code.

D. Harvesting and Software Rejuvenation

The expanded code includes predefined areas where changes can be made. To keep these changes from being lost when the application is expanded again, the expander can gather them and re-inject them when re-expanded. Gathering and putting back the changes is called harvesting and injection.

The application can be re-expanded for various reasons. For example, the code templates of the elements can be improved (fix bugs, make faster, etc.), new CCCs (add a new logging feature) can be included, or a technology change (use a new persistence framework) can be supported.

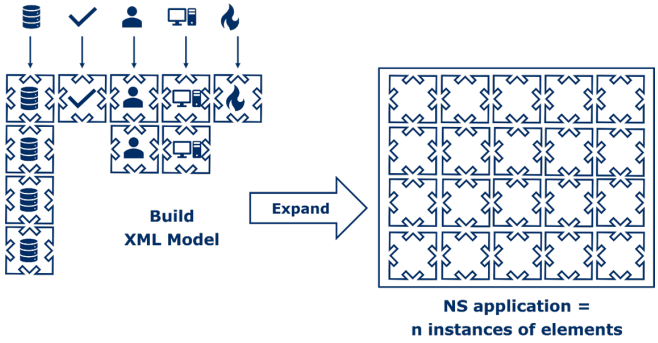

Fig. 2. Requirements expressed in an XML description file, used as input for element expansion.

Software rejuvenation aims to routinely perform harvesting and injection to ensure that constant enhancements to the element code templates are incorporated into the application.

Code expansion accounts for more than 80% of the application’s code. The expanded code can be called boilerplate code, but it is more complex than what is usually meant by that term because it deals with CCCs. Manually producing this code takes a lot of time. Using NS expansion, this time can now be spent on constantly improving the code templates, developing new templates that make the elements compatible with the latest technologies, and meticulously coding the business logic. Changes to the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. All developers can apply a modification to a code template by one developer across all their applications with minimal impact, thanks to the rejuvenation process (see Figure 3).

IV. RELATED WORK

While searching for relevant literature, we found some papers discussing Composable Architecture. The most relevant publications are, on the one hand, a paper about a possible

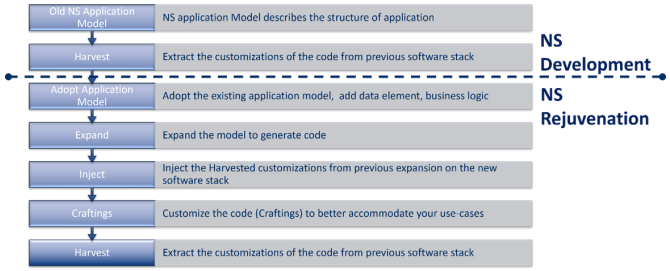

Fig. 3. NS development and rejuvenation.

methodology and demonstration by use case for implementing Composable Architecture, and on the other hand, the book of AW Sheers, "The Composable Enterprise" [9]. While Scheer [9] tried to rearchitect a complete organization, Ivas [10] provides a methodology to introduce Composable Architecture and demonstrates it with a use case. The paper also provides a good literature review, noting that only a few academic papers discuss Composable architecture (searches on Google Scholar, Web of Science, and Resource Gate). At the same time, thousands of papers are found in the professional literature (via Google), pointing out the need for academic attention to the subject. The paper proposes a methodology consisting of 6 steps (from [10]):

- Understand business drivers and objectives. The first step is to understand the business background of the initiatives, i.e., the rationale, purpose, and scope from the business point of view.
- Understand the holistic scope of the initiative. The second step is to understand what value stream steps and business capabilities from Holistic value delivery are affected by the initiative and how.
- Understand the current situation. Understand and sketch the scope of the current solution (architecture).
- Understand the situation and needs at the enterprise level. Identify if any components can be reused or optimised by this solution at the enterprise level or if there are other future initiatives with the same need.
- Design as API -first Headless PBC (preferably according to MACH [11]). If you need to implement a new service, you should preferably design it according to MACH principles. Otherwise, deliver business change by creating new or optimising existing monolith modules by API-first and Headless MACH principles.
- Implement business-IT aligned PBC solution and consolidate. Implement the agreed business-IT solution and consolidate any old solutions into the new solution that implements the same functionality (business capability).

For an Enterprise Architect, those steps are logical and provide excellent guidance. We argue that, next to the need for academic attention to applying Composable Architecture for (re)introducing functional re-uses, there is a need as well for academic attention to properly operationalizing/implementing Composable Architecture to make the dream of McIllroy a reality and not a nightmare.

For this extended paper, we sought additional work on the CAF. We knew that since the publication of the original paper, there were some conferences, mainly non-academic, that had the topic of CAF on the agenda. We can start by stating that, to the best of our knowledge, we have not found any additional academic research on CAF. We have found a lot of gray literature on the subject, indicating that the concept is gaining traction with practitioners and is becoming part of the heuristic body of knowledge of Enterprise Architecture.

For starters, we looked again at Gartner. In 2025 Gartner starts to associate the topic of composability with Digital Experience Platform (DXP), that focuses on the constant need to adjust to new user journeys to keep and attract customers. The link to the need to constantly adapt, change, and evolve is all variations on the same theme: you need a CAF. In the 2025 Magic Quadrant for DXP [12], Gartner states that by 2026, 70% of organizations will be required to adopt composable DXP technology (as opposed to monolithic suites) and explicitly defines that vendors must offer "modular, API-first approaches, with a set of discrete, task-oriented and independently deployable PBCs.

Secondly, we refer to the Mach Alliance, a consortium advocating Microservices, API-first, Cloud-native, Headless (MACH) architectures, heavily featured composability in 2025. They hosted "The Composable Conference 2025" (April 2025, Chicago) with hundreds of tech leaders, focusing on composable approaches and real-world MACH case studies. The alliance's insight articles note that composability has "come of age" – e.g., pointing out that "in 2025, composability has become a mandatory requirement" for DXPs and highlighting that many new DXP entrants adopt the PBC approach. The MACH Alliance also offers whitepapers on evaluating and integrating composable solutions, indicating an increasing level of practical guidance available in the industry [13].

Thirdly, we refer to SAP. In early 2025, SAP introduced the concept of Composable ERP, reframing its traditionally monolithic ERP suites into modular components. At the ES-ICONF 2025 conference [14], SAP's approach was detailed: they identify PBCs within ERP (e.g., Finance, Inventory, and Logistics as separate capabilities) and allow customers to implement them independently and integrate them as needed. SAP even defined composability similarly to Gartner, as "the ability to select, assemble, and rearrange components to adapt to specific and evolving requirements". What's significant is SAP's explicit breakdown of a PBC's internal architecture into three layers – User Interface, Application/Process Logic, and Unified Domain Data Model. This provides a more formal metamodel for PBCs in an ERP context, essentially an extension of CAF: it suggests that each PBC should include its own UI and business logic, and should share a standard data model where appropriate. SAP's messaging around Composable ERP positions it as the next evolution beyond post-modern ERP, aiming for flexibility and the ability to mix and match modules. This can be seen as an "alternative framework" in that it applies composability within a specific

enterprise context (ERP) rather than the general application composition of Gartner's CAF. But it's clearly inspired by the same principles and brings them to life in a real product strategy. For practitioners, the SAP case is a strong validation that composable architecture is not just for greenfield digital projects – it's influencing core enterprise systems now. (Other major software providers like Salesforce have similarly started promoting modular, "API-based" architectures in their ecosystems, effectively encouraging composable thinking within their product lines.)

We finish by mentioning that most of the ambiguities that will be discussed in the next section, also pop up in the gray literature (examples: [15], [16], [17], [18], [19]) and that there are ample posts to be found on the web that discuss the issues of defining PBCs, finding the right granularity and using the proper technologies. These topics will be addressed in Section V and will be shortly revisited in Section VI as well.

V. Critical Analyzis of Gartner's Reference Architecture for Composable Business Technology

In this section, we will take the components of the CAF (see Figure 1) and critically analyze them. We start by looking closely at business capabilities and how they should help align business and IT. We continue by discussing the modularity of PBCs and the different types defined by the CAF. We end this section by examining PBCs and CCCs and by delving into the requirements for a PBC platform.

A. Defining Business Capabilities

The concept of business capabilities originates in the resource-based view of companies [20], where it is considered vital to identify resources and capabilities that provide a competitive advantage [21]. This evolved into the idea that companies need to know "what" kind of activities they are undertaking, and gave rise to using the term business capabilities. Although decades have passed since the first mention of capabilities, there is no agreed-upon definition of business capabilities, nor is there consensus on how they should be defined, named, and used.

Table II gives an overview of 4 existing definitions of business capabilities. The first three are from industry sources, while the last is an academic source. There is some overlap, but each has specificities, as outlined in the key differentiators in the table. It is beyond the scope of this paper to provide an overarching definition of a business capability. We refer to [21] for a comprehensive literature overview on the subject. However, it does point to a weakness in the CAF — how to define something that lacks consensus?

Another fundamental issue with business capabilities is that they focus on the "what" rather than the "how". Formulated otherwise, they provide a black-box or a functional view. Traditional/heuristic functional design pays little attention to the fact that moving from function to construction (and vice versa) is not a straightforward, one-to-one mapping. It is an m-to-n mapping and requires a creative process that is not easily caught in an algorithm and requires architectural guidance. This has implications when a company wants to create its business capabilities map, starting from its current structure (how the company is organized). As there is no one-to-one mapping between functions and constructions (and vice versa), one ends up with multiple functions linked to a single construction, and vice versa. To make matters even more complicated, the TAO theory [22] argues that functionality is relative. An object, that is part of the construction, does not have functionality as a property. It only acquires functionality when a subject intends to use an object for a particular purpose. Functionality is thus a relation between subject and object. This implies that when different subjects (persons) build a functional architecture (using business capabilities), they are unlikely to arrive at the same result.

The cornerstone of Gartner's CAF is PBCs. However, as argued above, the definition of business capabilities can be problematic. If the definitions are unclear, then the implementation into PBC is suspect. For some industries, there are business capabilities frameworks such as BIAN [23] for the banking industry or NBility for energy transmission and distribution in the Netherlands [24]. There are companies like Leading Practice [25] that offer capability frameworks for some industries at a price. Still, many sectors lack such frameworks (openly or for pay).

As such, a necessary condition for using the CAF, being well-defined business capabilities, is already a tough nut to crack, and scientific guidance on how to do it is lacking. Some heuristic guidance will be provided in Section VII.

B. Business IT Alignment using PBCs

Gartner considered the business capabilities to be well-defined, shared between business and IT, and as an accurate alignment tool between the two. They refer to "Fusion Teams" as the secret formula to create this shared understanding. Fusion teams are teams that include both business and IT people. Close collaboration between the two will lead to a shared understanding and better solutions. This idea is also present in Agile Frameworks such as SAFe [26], where agile teams are considered multi-functional (mix of business and IT) and have people who know the job best close together, resulting in better designs than those imposed by intentional architecture. Or, stated otherwise, the PBCs are to be built bottom-up rather than defined top-down.

Existing heuristics for developing your business capabilities (see Section VII) treat them as part of the realm of strategy. They are to be aligned with the mission, vision, and strategic objectives. This implies that PBCs are to be built top-down. Such an approach will typically result in a set of business capabilities that are sufficiently understood by IT. The alignment between business and IT is at risk.

From NS, we know the importance of having an anthropomorphic design. This means that module names should reflect something that exists in the real world, as this increases understanding of the module's (intended) function. The same holds

TABLE II
COMPARISON OF BUSINESS CAPABILITY DEFINITIONS

Source	Definition Summary	Focus	Abstraction Level	Key Differentiator
OMG [27]	Ability or capacity a business possesses or exchanges to achieve a purpose or outcome.	Purpose-driven, exchange of capabilities	High-level, conceptual	Emphasizes exchange and specific purpose.
TOGAF [28]	Ability an organization, person, or system possesses; describes what can be done, not how.	Generic ability, independent of implementation	Very high-level, abstract	Stresses agnosticism to process or structure.
BIZBOK [29]	Ability needed to perform to achieve mission; expressed in terms of outcomes, independent of structure/process.	Mission alignment, outcome orientation	High-level, linked to mission	Highlights mission-critical nature and outcome focus.
Academic [21]	Capabilities link strategy (why) and implementation (how); describe what activities must be performed to achieve objectives, independent of processes or structure.	Strategic alignment, activity-based view	High-level, bridging strategy and execution	Connects strategy to execution explicitly.

for business capabilities. If they are insufficiently fine-grained and abstract, they no longer represent business reality. Still, if they are too detailed, the number of business capabilities is considered too large and thus complex.

C. PBCs and Modularity

Gartner's PBCs are modular, where modularity is defined as "partitioned" into a cohesive set of components [3]. Garter further adds that *"The granularity of PBCs, as with all modular systems, is a common design challenge. Modular components that are too large may be easier to manage, but they are harder to change are use in new compositions. Components that are too small may be easier to assemble but harder to isolate, identify, find or change."* [3]. This statement is vague. What are the objective criteria for being too large and too small? Secondly, as neither is considered a good modular design, what are the requirements for a sound module size? Some of the gray literature mentioned in Section IV considers the PBC to be the optimal size, positioned between the monolith and the microservice. We could dedicate a paper to the discussion of microservice size, but the issue is that an objective criterion is missing.

NS theory explicitly defines the optimal module size to facilitate anticipated changes. A module must be split into smaller modules until each complies with the four NS theorems: SoC, AvT, DvT, and SoS. When all the theorems are met, the optimal size is reached. As these are the necessary conditions for system stability under change, violating them will introduce Combinatorial Effects (CE).

Separation of Concern (SoC) teaches that each PBC should encapsulate a separate change driver. PBCs must be defined as fine-grained and very specific. For example, a possible PBC is Asset Monitoring. The asset could be an IT system (servers, databases) or an OT system (Operational Technology as SCADAs, Turbines, etc.). Although both are assets, secure monitoring for both requires different implementations. This would mean that if we used only one PBC for this, we would need two versions: one for IT and one for OT. It suffices to extend this way of thinking and conclude that insufficiently fine-grained PBCs would lead to multiple versions of the PBC and break the anthropomorphic relation between what something is (a PBC) and how something is done. Such an anthropomorphic relation is a required condition for a PBC platform as PBCs must be easily discoverable (see Section II)

Action version Transparency (AvT) enforces interface stability when the implementation changes. The CAF does not explicitly mention interface stability when the PBC implementation changes. However, it does note the need for technologies that result in loosely coupled systems, such as providing interfaces via APIs and event buses, and using technologies promoted by the MACH Alliance [11]. We will return to this point when discussing the PBC platform. However, AvT should not be limited to technical protocols and implementations. PBC definitions should also address semantic issues related to version transparency. In case a business capability introduces a new feature, a default behavior needs to be defined in case the new feature is not specified.

Data version Transparency (DvT) ensures that changes to data attributes do not affect processing functions that do not use those new attributes. The CAF does not mention this concept; it is left to the data fabric. This paper will not discuss the vagueness or evolvability of concepts such as data fabric.

Separation of State (SoS) required all modules to maintain state and make it externally visible. The CAF does not mention the need to state-keep the PBCs. Similar to AvT, MACH technologies are cited for promoting the use of process Choreography over process orchestration for evolvability. Independent of whether this is valid, choreography (and orchestration) requires state-keeping and exposition to trace process issues back to the failure location. Similar to AvT, statekeeping should also be looked at semantically, i.e., what is the actual business state at specific points in the process flows.

D. PBC Types

Garter introduces PBC Types, such as application PBC, data entry PBC, analytics PBC, Digital Twin, and Process. What is missing is the relation between a PBC and a PBC Type. Is a PBC comprised of multiple PBC types, or do they have a one-to-one relationship? The latter would be strange as to do something, a "what" or a capability, you require things to make it happen, "hows", and the PBC Types are pointing more into the direction of a "how" than in the direction of a "what". The PBC Types have similarities with NS elements.

- Application PBC Type (action) versus the Task Element

- Data Entity PBC Type (reference) versus Data Element
- Process PBC Type (flow) vs Flow Element

For NS, an Analytics PBC would be an Application PBC, where the action/task is to perform some analytic operation. A similar reasoning applies to the Digital Twin (thing) PBC, where this would combine data and task elements in NS. We notice Garter's difficulty in addressing concepts at their suitable granularity and abstraction level.

E. PBCs and CCCs

NS recognize that CCCs must be treated as any other change driver and require splitting into different modules and proper encapsulation. PBC focus on business, not technology. The technologies used to implement, run, and deploy the PBC are abstracted, and the only type of guidance regarding technology is to use MARCH-compliant technologies. As stated in the previous subsection, we notice an analogy between PBC Types and NS Elements. NS Elements are there to allow proper encapsulation and separation of CCCs. One would expect something similar in PBCs, but this is not the case. This represents a lack of design criteria for the PBC and could result in proper SoC regarding capabilities/functionalities but zero evolvability for the cross-cutting concern and associated technologies. If CCCs aren't modularized, a PBC might be internally entangled with specific technologies or policies, making it non-evolvable when those concerns change. For example, if every PBC implements its own logging or security, a change in security policy would require modifying many PBCs (a combinatorial effect). As new technological implementations change faster and faster, ignoring these change drivers will introduce CEs. Gartner implies leaving this to the PBC Platform.

F. PBC Platform

Much of the PBC implementation magic is left over to the PBC Platform. It must facilitate the discovery of the available PBC, the composition of PBCs and the deployment of PBCs. We already argued that proper discovery requires anthropomorphising the PBC namespace and its associated granularity to avoid PBC versioning without meaning. In the previous subsection, we argued that Gartner seems to push the treatment of the CCCs toward the PBC Platform. Instead of addressing CCCs at the PBC level, they need to be discussed at the PBC Platform level. This would be a possibility were it not for the fact that, as a selection criterion for a PBC platform, it is currently not mentioned in the CAF

Another crucial aspect of the PBC platform is the design, deployment and running of the PBCs. Gartner sees that a change in a PBC Type would yield an update of all composed applications that use such PBC Type. In NS, updates to the templates that make up the elements are handled through expansion and rejuvenation. The new version of the element template triggers a rejuvenation cycle, updating all instances where it is used. The final step is to deploy the application.

Gartner simplifies this concept with an example of Planning PBC used in two compositions [5]. The update to that Planning PBC would trigger updates to both compositions. The question is, what is being updated? Is it a PBC Type that underlies the Planning PBC? Is it the Planning PBC template, or the code that makes up the PBC? Does this update result in one deployment of the Planning PBC shared by two compositions, or are there two deployments of the Planning PBC? In the former case, it would mean that the Planning PBC has zero customisations (even in terms of low and no-code). Otherwise, it cannot be used in two different composite applications without putting extra differentiation logic into the Planning PBC or the composite applications. In the latter case, it would mean that running instances of Planning PBCs are not identical, and the difference must be managed on the Platform, resulting in different PBC versions. We already discussed the issues related to that topic.

The problem is that there is no clear guidance on how the PBC platform should handle this. The PBC Platform must also conform to the four NS principles, or it will not allow the evolvability Gartner portrayed in its CAF.

VI. Validation By Focus Group

The previous section looked at the components of the CAF and tried to point out possible operationalization issues employing NS and business capabilities theories. To avoid our findings being perceived as overly opinionated, we conducted a small experiment with students at the Antwerp Management School who had just been exposed to NS. The idea was to investigate how they view a concept such as Composable Architecture after learning about NS.

Between September 2024 and November 2024, the Master in Enterprise IT Architecture (MEITA) students of the Antwerp Management School were exposed to NS theory for sixteen hours. The MEITA is an executive master, and students in the program already have many years of practical experience in IT Architecture. At the end of this period, they were given the Gartner paper on the CAF and asked to read it using their newly acquired knowledge of NS. All students received the reading material beforehand, were asked to read it, discuss it in groups for 30 minutes, and then provide a summary of their findings in 5 minutes. In total, 18 students participated, split into four groups.

The first group had difficulty understanding the meaning of PBCs. They realized that the approach can only succeed if business and IT people define the PBCs. They expected more guidance on how to do this and foresaw evolvability issues, as no mechanisms were foreseen to adequately address SoC beyond business changes (what about technological changes?).

The second group tried to list the pros and cons of the framework. They found the use of no- or low-code with a marketplace of PBCs to be powerful. They considered the recommended use of headless promotion technologies to be a good way to decouple UI and logic, while adhering to SoC at least for those two concerns. Group one also struggled with understanding PBCs and claimed they had never observed it in their practices. They see the framework's application more in classifying your existing applications according to business

capabilities, and then use the Composable Architecture Integration platform to recombine existing applications.

The third group also struggled to understand PBCs. They compared the main characteristics of PBCs with the NS theorems. Modularity resonates with NS, but the framework struggles to describe the level of granularity. They see SoC applied to some extent but insufficient to be NS compliant. They saw that the second characteristic, autonomy, would require SoS, but this is not mentioned. They make a similar remark about PBC orchestration. They conclude that the fourth characteristic, discoverability, fails to account for the need for AvT and DvT.

The fourth group noticed that while NS is about change over time and minimizing ripple effects, PBC is about building fast. The characteristics of PBC are such that nobody in their right mind would want them, but the framework is insufficiently explained on how to do it. On the other hand, NS tells you how to do it, and doing it the NS way requires significant effort. The group also identifies the PBC granularity uncertainty as a source of future issues. For instance, the sales business capability, packaged in a PBC, may or may not answer to the different sales business capabilities needs in large and complex organizations, opening the door for violation of the four NS principles quicker than expected.

The different groups struggle with similar issues. Once exposed to NS, one can ask more profound questions about how implementation should occur and whether a proposed solution has the characteristics it claims to have. It is important to note that this does not necessarily mean that NS is the optimal solution to operationalize the concept of PBC, as only a single methodology was evaluated in the focus group. Nevertheless, it does seem to validate that there is a need for a strategy to operationalize a PBC architecture through more concrete guidance, such as provided to a certain extent by NS.

The focus group reports align with our findings from the additional literature review, as discussed in Section IV. As CAF becomes more widely known, practitioners start to notice the gray areas and try to fill them with heuristic knowledge that often is a practical application of the NS concepts. NS never claimed to have found something new, but it does claim to have a theoretical and scientific method for unifying and grounding existing heuristic knowledge.

The same experiment was conducted in November 2025 with the new MEITA student cohort. Three groups of three students were formed and were asked to analyse CAF with NS. The reported issues are similar to those found by the previous cohort.

VII. Extending the CAF

In the previous section, we pointed out several shortcomings of the CAF. We will now try to guide you on how to overcome these shortcomings. Wherever possible, we will provide direction supported by science, though this will not always be possible. We will start with guidance on the definition of business capabilities, followed by the size of PBC modules. We continue with PBC Types guidance, address CCCs, governance of the PBC platform, and end with an overview of our advice.

A. Defining PBCs

Before we can package the business capabilities, we need to define them. In Section V-A, we have seen that a common understanding of what a business capability is and how it should be mapped remains lacking.

Innocom, an enterprise architecture consultancy and education institution, offers a two-day seminar on PBCs. In their approach, participants are asked to identify the essential tasks in a given context — the top tasks. They refer to the book "Top Tasks: A how-to guide" by G. McGovern [30], that helps distinguish top tasks from small ones. Those top tasks are good candidates for being packaged. If those top tasks are considered your top required business capabilities, listing them will provide you with the required PBCs.

We have seen in Section V-A that THE functional view, and thus the capabilities view, of an enterprise does not exist. As such, we might as well try to find one that is as close to reality as possible using the top tasks method. The results of such a bottom-up approach may be validated/adjusted/augmented by top-down approaches. There may exist a publicly/privately available business capabilities map that can serve as validation or a challenging framework for your top tasks. The company may have a strategic map linking mission, vision, strategy, and required capabilities. Both types of business capability maps risk being abstract and disconnected from the nomenclature used by business people in their day-to-day work. Abstraction is suitable for general usability and re-use, but it can hamper adoption and buy-in. We recommend finding a middle ground that connects the abstract and the concrete, and using anthropomorphic naming for the identified business capabilities to ensure maximum unambiguous understanding.

One could use an advanced enterprise engineering method, such as DEMO [22], that is backed by science, to construct the enterprise ontology and, from there, enforce a one-to-one mapping to a set of enterprise functions/capabilities provided by the identified ontological transactions. The ontological transactions could be used as a business capability model because they are outcome-focused (a key feature according to some definitions of business capabilities (see Section V-A)).

Gartner recommends forming fusion teams to ensure both business and IT are at the table and to foster a common understanding. This is good advice. We recommend adding people with a strong methodological background who can distinguish and steer discussions away from technological implementation, and who can reformulate and generalize concepts raised during the interactive workshops among all relevant parties. Such profiles will help in the convergence between bottom-up and top-down approaches.

B. PBCs modularity and granularity

PBCs are considered modules. The raison d'être of PBC is to have a composable architecture that allows combining,

recombining, and changing PBC to create new applications. The PBCs must favour evolvability.

As NS is a theory of modularity, it provides guidance on how to create and size your modules to enable evolvability. Recall that a system is considered evolvable if it is free of combinatorial effects. A combinatorial effect (CE) occurs when a bounded functional change results in a mount of work that is proportional to the size of the system. The necessary condition for a modular structure to be free of CE is given in the 4 NS theorems. In what follows, we will translate NS into the PBC context and demonstrate the need for compliance through an example.

1) Separation of Concern: The first NS theorem—Separation of Concerns (SoC)—states that a module should encapsulate only one concern, where a concern is defined as a change driver. As such, a PBC should encapsulate only one concern. If multiple concerns are combined in a PBC, than those concerns would need to change all together as a group and not individually, and that would basically mean that the change driver can only exist at the PBC level, not in its components. Our advice is two-fold: either you create PBCs that respect SoC, or you do not allow variations in a PBC.

Let us elaborate by means of an example. The ”invoicing” business capability is part of every company. It makes sense to create a PBC for invoicing that is used across the company to receive payment for delivered services. But what do we mean by invoicing? Is it only the creation of the invoice in electronic format that states the amount to be paid, or do we also include sending the invoice via e-mail, an e-invoicing system, or physical printing and sending? Invoicing risks to contain multiple concerns. NS tells us to split the PBC into smaller PBCs: invoice_calculation, invoice_e-sending, invoice_e-mailing, invoice_printing, invoice_physical-sending. If the PBC is not divided into these smaller modules, CE will be introduced in the landscape. Let us assume that a company offer five services, each delivered by a separate branch. Suppose they all use the PBC ”invoicing”. One of the branches uses the PBC invoicing to calculate and e-send the invoice, another only to calculate and print. Different branches use different concerns packed in the same PBC. Suppose now that a new version of e-sending is required (that may be country-specific). But not all branches require this latest version, and by introducing a bounded functional change, we impact all parts of the organization that use the PBC, meaning we introduce a CE. One could argue that this could be solved by integrating into the PBC the two different versions of e-sending, where the original one stays untouched, and the new one is added, thus being compliant with the Open/Close principle of Robert Martin: modules are open for extension, closed for modification [31]. Suppose now that the company acquires another company to be integrated as a sixth branch, and it needs to set up its invoicing with the invoicing PBC. As the PBC is a package that hides its insides, how will it know which e-invoicing procedure to use? The insides of the PBC need to be known to allow implementation. This contradicts the idea of a module acting as a black box and the concepts of packaging. Combining multiple concerns into the invoicing PBC would mean that all branches need to use the ”invoicing” PBC and will use it in the same way: first calculate, then e-mail, then e-invoice, then print, and finally physically send. You use the full functionality of what is defined as invoicing, or you cannot use it. The moment there is a need for a different use of that PBC, there are various change drivers, and they must be split off. By splitting, we really mean the creation of a PBC to address that concern (like PBC ”e-sending_europe and ”e-sending_india”), not create a new version of the original PBC ”Invoicing V2” that will include the new version of e-sending for India.

2) Separation of State: The second NS theorem is Separation of State (SoS). It postulates that each module should maintain its own state and expose it. As PBCs are modules, they should also preserve and expose their state. Fine-grained PBCs enable more detailed state tracking and actions based on those states. While coarse-grained PBCs hide the orchestration of their inner workings from the external world, fine-grained PBCs require an externally visible orchestration PBC to allow partial execution of the smaller PBCs.

Besides the default requirement that a PBC maintain state, we advise treating the PBC as an atomic transaction and asking whether it can be treated that way. If not, splitting the PBC becomes a necessity.

We would also like to warn of potential Babylonia confusion of the word ”state”. At the technology level, the default standard is stateless APIs. PBCs will be accessible via APIs. As such, some may think that the need for stateful PBCs is opposed to the usage of stateless APIs. The meaning of ”state” is different and has little to do with the other. When working with fusion teams, it might be necessary to skip the discussion of the significance of the word ”state” asap to avoid further misalignment.

In our invoicing PBC example, the module should have states as ”invoice being calculated”, ”invoice calculation error - error description: xxx”. In the previous subsection, we advised either using a single PBC or splitting it into fine-grained PBCs. When the PBC is kept as a whole, a grouping of tasks and data, it should expose state about that whole rather than its inner workings. Issues about the inner workings should be part of logging, of course, for troubleshooting purposes, but as the inner tasks of the PBC are not allowed to be used, they cannot be re-called or acted upon. If the coarse-grained invoicing PBC had a state of ”error sending e-invoice”, the entire ”invoicing” transaction must be rolled back. If we had fine-grained PBCs, the invoice_calculation, invoice_e-mailing, invoice_printing and invoicing_physical-sending could be ok, but the invoice_e-sending could have failed, only requiring a rollback of that transaction. This should make it clear that going for the coarse-grained PBC, if SoC is ignored, SoS gives you an additional sanity check - are you really sure you want to do this, because you will now have to solve rollback issues, complexifying the PBC.

3) Action Version Transparency: The third NS theorem is Action version Transparency (AvT). It postulates that an internal change to the implementation of a module, assuming iso functionality, should not impact the calling module.

Applying this theorem to PBC is straightforward: all available interfaces to the PBC must remain the same when internal upgrades to the PBC are performed. Examples of internal upgrades that should not affect the interface include performance improvements, bug fixes, and changes in the underlying technology. To be even more precise, the API URLs should not change.

Changes in functionality can break the interface. If this is the case, the PBC should no longer remain the same PBC; a new one must be created. The new PBC should have a different anthropomorphic name that reflects its new function and exist alongside the previous PBC. Users of the old functionality are not affected, and those requiring the new functionality connect to the new PBC. Suppose the first functionality is to be replaced over time. In that case, an orchestrated transition from the old to the new can occur—a gradual migration to the new URL, potentially facilitated by a platform rejuvenation facility (see further).

Our advice regarding AvT is to adopt a clear versioning strategy that distinguishes between internal and external versioning. Internal versioning should not impact the PBC; external versioning is subject to migration effort that can be proportional to the system's size, but this can be resolved through automation (see further).

Let's apply this to our Invoice PBC example. Changes to a new mailing system, a more efficient invoice calculation method, a new printing system, or a new e-invoice system should not affect the functioning of the PBC, nor do they justify a new version (with a new name). Only the internal versioning is updated. Suppose a new e-invoicing system has a different interface from the current one, and that this change breaks the interface. The internal change ripples through to the PBC. At this point, we have identified a change driver, and Soc tells us we should split off this module. Maintaining AvT also pushed toward finer granularity. If we do not want to create a new separate PBC dedicated to e-invoicing, then we must create a new version of the invoicing PBC. That new version should have a new name, like "invoicing with new e-invoicing," to indicate the functionality difference from the original. Note that such a solution will require maintaining two PBC that share the same code for invoicing_calculate, invoicing_print, invoicing_e-mail, and invoice_physical-sending. At this point, we start introducing the ripple effect and even CE when future changes to those PBC internals are required. Unless the PBCs are an aggregation of more fine-grained PBCs that the business cannot access (see further).

4) Data Version Transparency: The fourth and last NS theorem is Data version Transparency (DvT). It postulates that adding information to the data structure passed to a processing function should not affect the processing function.

PBCs will perform actions on data (see Section VII-C) that will be passed to them via APIs or messages. The data transmission architecture or directive must favour DvT. This means that in a distributed environment, we favour sending entire objects (packed in JSON or XML, for instance) (aka stamp coupling) rather than passing individual attributes (aka data coupling). In an in-memory setup, this means objects are passed by reference, not by value. Security professionals will typically oppose this, as disclosing too much information may lead to misuse or leakage. However, if the PBC are adequately implemented, with security as being part of their design, then this should not be a problem as there are sufficient methods available to shield, anonymize and validate data access authorization. This does imply that security must be set at a fine-grained level. It is beyond the scope of this paper to discuss data security, but from an evolvability perspective, pass-by-reference is superior to pass-by-value.

Our advice regarding DvT is to ensure a data transmission architecture is put in place that supports secure stamp coupling for PBC-to-PBC communication.

C. PBC Types

According to the CAF, PBCs are of a specific type. Gartner is unclear whether they mean there should be a one-to-one mapping between a PBC and a PBC type, or whether they suggest that a PBC should be built with sub-modules of a specific type. Those sub-modules are no longer PBCs, they do not address a business capability but rather a generic task or thing.

NS introduces the notion of elements, the elementary building blocks an application is made off. Whether an application represents the implementation of a single business capability or a collection of business capabilities working together, it will be built using elements. Elements define the basic operations performed by an IT system: represent data, perform tasks on data, execute tasks in a specified order (orchestration/choreography), connect to other elements, and trigger the actions an element is supposed to perform. In our opinion, Gartner makes the PBC approach overly complicated by introducing PBC types. It suffices to say that PBC should be made out of smaller modules of a specific type/pattern/element. As those types/patterns/elements are generic by nature, they no longer represent a particular business capability.

Our advice is to drop the PBC Types and to see PBCs as a composition of generic data, action, flow, trigger and connection patterns/elements. Those patterns/elements should be highly standardised and also be compliant with the NS principles. We will return to this point in the section on PBC platform governance.

In our example, what type of PBC should we assign to our invoice PBC? The PBC will contain data and the actions taken on it. If the PBC then Data PBC Type or an Application Action PBC Type? And if it is the combination of the two, does that then mean that the invoice PBC is a combination of

an invoice Data PBC type and a calculate invoice Application Action PBC Type, etc.? These questions move us away from what is essential (the functionality of the PBC). They can be shortcut by treating all PBC as composed of data, action, flow, trigger, and connection patterns/elements.

D. PBC and CCCCs

In the previous section, we considered PBCs as composed of elementary patterns implemented in technology. To protect against technological evolution, that cuts across all implementations, we need to ensure that technological concerns or change drivers are adequately separated from the core elementary functionality. Elementary patterns need to be carefully designed for this. If not, technological changes will impact all patterns, PBCs and PBC instantiations.

There can also be CCC at the PBC level. A classic example is security. Our invoicing PBC needs, by design, to include security. Only those authorized to use the PBC should be allowed to do so. All PBCs must incorporate this concern by design. This will result in each PBC having some elementary patterns associated with it to take care of authorization. And those elementary patterns must have the technological aspect of authorization properly separated.

Our advice is to look for CCC at the PBC level, and security must be there by default. If not, the security-by-design or zero-trust principle erodes from the top.

E. PBC Platform Governance

According to Gartner's CAF, the PBC platform should handle discovery, composition (via low-code/no-code), and deployment of PBC. We will handle them one by one.

1) Discovery: In previous sections, we motivated the usage of anthropomorphic naming for the PBCs. Next to naming the PBCs, there should also be a hierarchy of PBCs that allow the creation of a name space. From public and privately available business capability frameworks, we know that business capabilities are organized in a hierarchy. We advise using an anthropomorphic hierarchical name space to classify business capabilities and PBCs.

Returning to our invoicing example. Within the company of one of the authors, invoicing is present across multiple business capabilities - see Figure 4. There are three invoicing business capabilities: one for certificates, one for commodities and one for services. Note that those BCs aggregate billing and invoicing. Although related, they differ in the actions required. If we follow our own recommendations, billing and invoicing should be split at the second level of the Sales Management BC into "Billing Management" and "Invoice Management", and level three should be split accordingly (see Figure 5). We could now have three PBCs: one for certificate invoicing, one for commodity invoicing, and one for service invoicing, or we could assign to those three BCs, only one PBC, being our invoicing PBC. From a Business Capabilities Framework perspective, it makes sense to state that we bill for three reasons clearly. Still, at the implementation level, a single set of tasks must be performed. This leads us to the advice that the Business Capabilities Framework must be part of the PBC platform and should serve as a path toward a PBC, but not as the list of PBCs themselves. Some BCs might not even require a PBC.

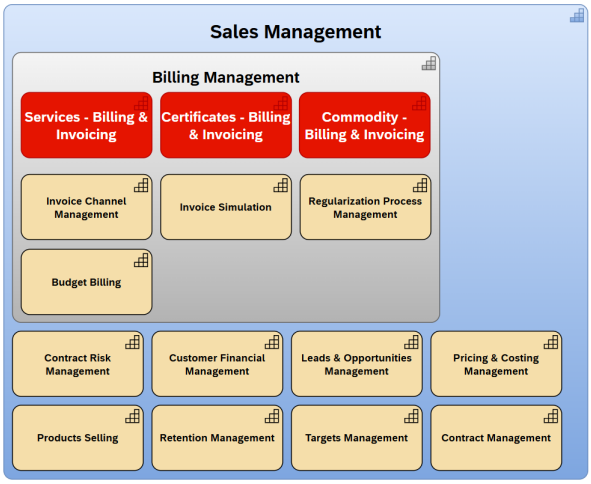

Fig. 4. Business Capabilities Framework - invoicing

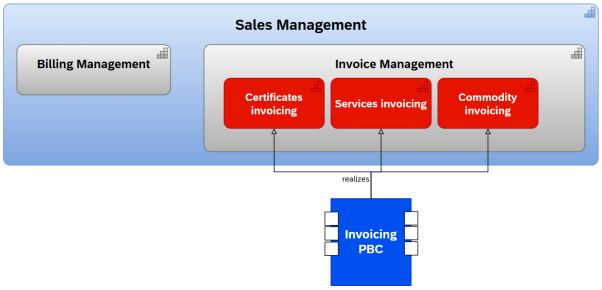

Fig. 5. Linking the Business Capabilities Framework to PBCs

2) Composition: To facilitate the reconstruction of PBCs in Composed Applications (CA), Gartners proposes a low-code/no-code platform. Although low-code/No-code is in line with McIllroys dream, we warn for the vendor/technology lock-in they represent. Migrating from one low-no-code platform to another will be a combinatorial effect. As those platforms are not open source, it isn't easy to assess whether they respect NS principles. We advise looking for and selecting low-code/no-code platform constructs that allow breaking PBCs into elements (data, task, trigger, flow, and connect), make PBCs with them, combine the PBCs into CAs and governing the platform accordingly. The platform may offer you appealing packaged features, but these may introduce CEs. Technological features often lure you into using them. It's not because something is technically possible or the technology is available to do so that one must do it. We are even more in favour of working according to the NS development

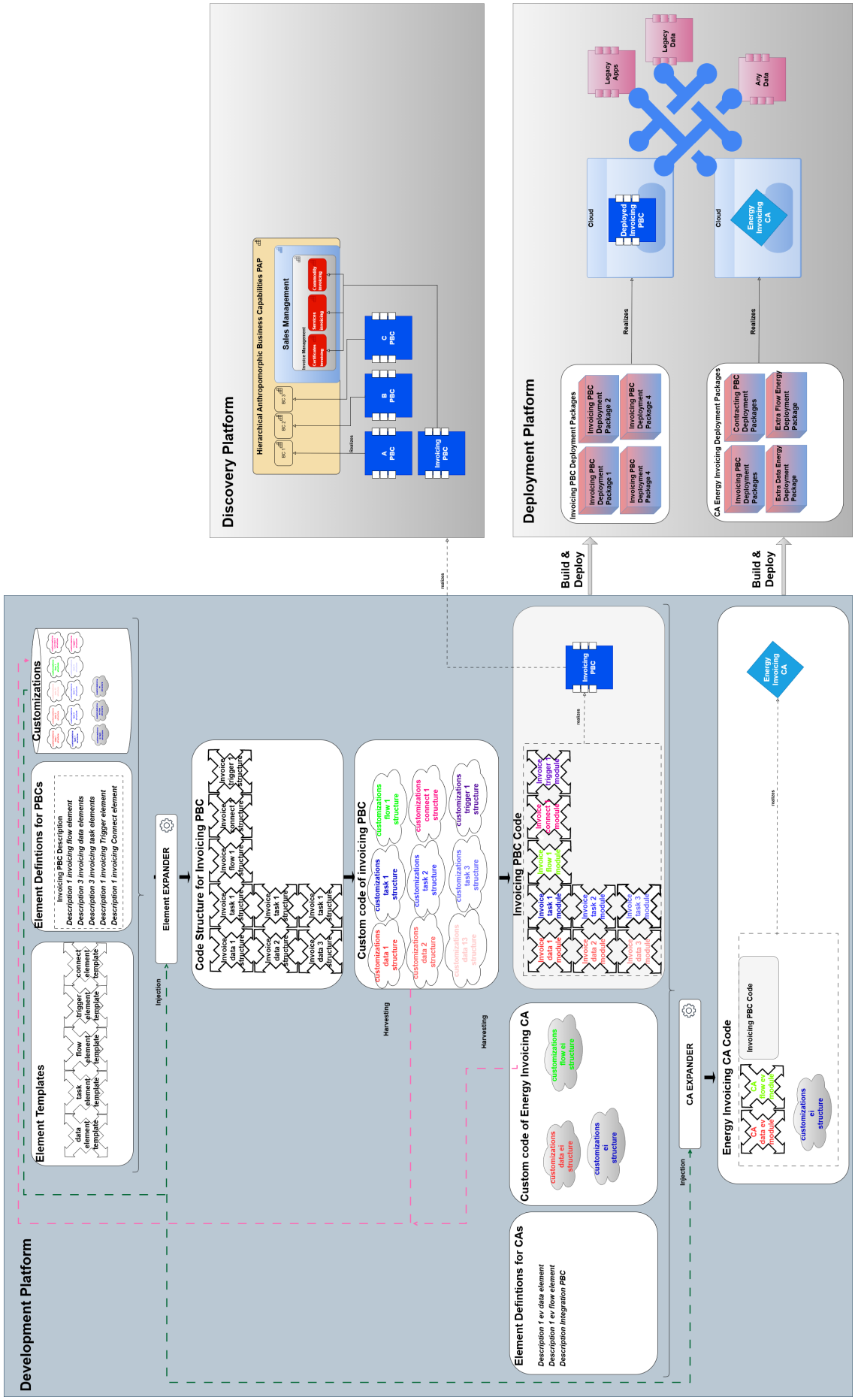

Fig. 6. Proposed PBC platform.

TABLE III
RECOMMENDATIONS TO IMPROVE GARTNER'S CAF

Area of Improvement	Recommendation Summary	Key Actions / Guidance
Defining Business Capabilities	Use a combination of bottom-up (top tasks) and top-down (strategic maps, frameworks) approaches. Employ anthropomorphic naming for clarity and adoption.	• Identify essential tasks ("top tasks") as candidate business capabilities. • Validate/augment with strategic maps or industry frameworks. • Use DEMO to identify essential transactions and capabilities. • Use names that reflect real-world business language for better understanding.
Granularity of the Modular PBCs	Use NS evolvability theorems	• Apply SoC, SoS, AvT and Dvt
Separation of Convern (SoC)	Ensure each Packaged Business Capability (PBC) encapsulates only one concern (change driver). Split PBCs as needed to avoid combinatorial effects.	• Split PBCs until each addresses a single concern. • Avoid variations within a PBC; create new PBCs for different usages. • Treat PBCs as atomic transactions when possible.
Separation of State (5oS)	Each PBC should maintain and expose its own state. Fine-grained PBCs enable better state tracking and rollback.	• Design PBCs to track and expose relevant states. • Prefer fine-grained PBCs for detailed state management. • Clarify "state" meaning to avoid confusion between business and technical contexts.
Action Version Transparency (AvT)	Internal changes to a PBC should not affect its interface. Use clear versioning strategies.	• Maintain stable APIs for PBCs. • Distinguish between internal and external versioning. • Create new PBCs (with new names) for breaking changes, enabling gradual migration.
Data Version Transparency (DvT)	Data transmission between PBCs should favor "stamp coupling" (passing whole objects) for evolvability.	• Implement secure, object-based data exchange (e.g., JSON/XML). • Ensure security at a fine-grained level. • Prefer pass-by-reference over pass-by-value for evolvability.
PBC Types	Drop rigid PBC types; instead, compose PBCs from standardized, generic elements (data, action, flow, trigger, connector) compliant with NS principles.	• Build PBCs from NS-compliant elements. • Avoid unnecessary complexity from PBC type distinctions. • Standardize patterns/elements for reusability.
CCCs	Identify and modularize CCCs (e.g., security) at both the PBC and element level. Ensure technological concerns are separated from core functionality.	• Integrate CCCs (like security) by design in every PBC. • Separate technological aspects from business logic. • Use elementary patterns for CCCs.
PBC Platform Governance	Enhance platform features for discovery, composition, deployment, and data integration. Apply NS principles to platform design.	• Use hierarchical, anthropomorphic naming for PBCs. • Link business capability frameworks to PBCs. • Select low-code/no-code platforms that support NS elements. • Apply NS principles to deployment and data integration. • Avoid vendor lock-in and ensure platform evolvability.

methodology. The basis for this are the NS elements, that are made up of code templates that can be instantiated into boilerplate code via expansion. To decide what data, task, flow, trigger, and connection elements you require, we recommend starting by creating a data model tailored to the PBC building blocks you need. This does not differ from Evans' advice in Domain-Driven Design (DDD) [32]: start by creating a bounded context. Next to the data, the task that will work on that data and the orchestration of those tasks. For this, state-machine models are recommended. The boilerplate code can now be customised, and those customisations can be separated from the boilerplate using harvesting, allowing them to be reinjected when the templates are updated—the rejuvenation process. The resulting library of PBCs is to be linked to the discovery features — the list of BCs and their corresponding PBCs. To create CAs, a set of PBCs and elements that facilitate integration and orchestration will need to be expanded and packaged. Again, a data model and a state machine can be used to represent the CA's logic. We propose a CA-expander that integrates, orchestrates (flows), and connects to PBCs.

3) Deployment: Thanks to build frameworks, the cloud, and Infrastructure as Code, deployment has become a less

painful process than in the past. The application of deployment pipelines and the organisation of teams according to the DevOps philosophy [33] is now a standard in the organisation and implementation of IT Operations. We want to point out that, of course, building your own deployment infrastructure or using a platform for it is, in itself, a vital change driver, and again, NS principles should be applied to facilitate deployment evolvability. We refer to [34] for more information on the subject.

4) Data Fabric: Regarding the Data Fabric component of the CAF, we give the same advice as for the PBC Types: ignore it. When the PBC needs data it can tap into that data via data elements. The data elements ensure proper encapsulation. Any data-providing platform, be it a simple DB or an advanced/abstract data Fabric/Lake/Lakehouse (whatever term is used), should respect the NS principles so that changes do not ripple through to the PBCs. If this cannot be guaranteed by the data provided, the data should not be integrated in the PBC, as they will hinder evolvability

F. Overview

Table III provides an overview of the recommendations we have put forward in the previous sections. Figure 6 gives a graphical overview of our proposed enhanced CAF. On the left side of Figure 6 you will notice the "Development Platform". It contains the different "Element Templates" (data, task, flow, trigger, connection, and other template types can be added) and the "Element Definitions for a PBC", containing the parameterization of the element templates. The combination of both goes into the "Element Expander" and results in the code structures (the boilerplate) code of a PBC. These code structures are customized and all customizations are harvested and put in the "Customizations" repository. During rejuventation (updates of the "Element templates"), the "Element Expander" will inject the customizations in the boilerplate code. The result is the PBC code. CAs are a combination of PBCs and some glue or custom coding. This is shown at the bottom left, where the "CA Expander" combines extra elements and existing PBC code into CAs. Again, all customizations are harvested, and reinjection can happen during rejuvenation. To the right, you will note the "Discovery Platform", that will link the available PBCs to the BC framework. At the bottom right, the "Deployment Platform" is shown. It contains the tooling to package the PBCs and CAs code into deployable units that end up in the cloud, where they can interact with all other applications (PBCs, CAs, and legacy).

VIII. Conclusion

Application re-use is not just a long-forgotten dream of McIllroy, but a focus of many companies. The ability to reuse and recombine applications to support the changing business conditions is an expectation many CEOs have of their CIOs. In recent years, the focus on functional reuse has been pushed aside by technology-focused integration patterns. Gartner puts functional re-use back on the map with its CAF, where PBCs are the essential building blocks of application landscapes. By using NS as an instrument of design and evaluation method for evolvable systems, we pointed out operationalization issues one might face when trying to implement the CAF, or one might use it to evaluate providers of solutions based on it critically. A focus group was used to validate and balance our findings. In this extended edition of our original paper, we convert our criticism into recommendations and improvements to the CAF, combining knowledge of NS and some heuristics, to arrive at a proven CAF for evolvable PBCs.

Acknowledgment

The authors thank Rudy Claes of Innocom for introducing them to Gartner's Composable Architecture Framework and conducting a brainstorming session about the framework and its evolvability. We also thank the Master in Enterprise IT Architecture (MEITA) students at Antwerp Management School (AMS - cohort 2024 and 2025) for their contribution as focus group.

References

[1] G. Haerens and H. Mannaert, "On the operationalization of composable architecture by means of NS theory," In PATTERNS 2025: The Seventeenth International Conference on Pervasive Patterns and Applications, pp 23-30, 2025.

[2] G. Coulouris, J. Dollimore, and T. Kindberg, "Distributed Systems: Concepts and Design Edition 3," ISBN:978-0-201-61918-8, 2001.

[3] J. Sun and Y. Natis. "Use Gartner's Reference Model to Deliver Intelligent Composable Business Applications," Gartner, ID G00720701, 2020 - refresh 2022.

[4] H. Mannaert, P. De Bruyn, and J. Verelst, "On the interconnection of cross-cutting concerns within hierarchical modular architectures," IEEE Transactions on Engineering Management, Vol. 69, pp. 3276-3291 2020.

[5] Y. Natis and G. Alvarez, "How to Implement Composable Technology with PBCs," Gartner, ID G00751018, 2021.

[6] H. Mannaert, J. Verelst, and P. De Bruyn, "Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design," ISBN 978-90-77160-09-1, Koppa, 2016.

[7] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," Science of Computer Programming, Volume 76, Issue 12, pp. 1210-1222, 2011.

[8] P. Huysmans, J. Verelst, H. Mannaert, and A. Oost, "Integrating information systems using normalized systems theory: four case studies," In IEEE 17th Conference on Business Informatics, Volume 1, pp. 173-180, 2015.

[9] A.W. Scheer, "The Composable Enterprise: Agile, Flexible, Innovative: A Gamechanger for Organisations, Digitisation and Business Software," ISBN:978-3-658-42482-4, Springer, 2024.

[10] I. Ivas, "Implementation of Composable Enterprise in an Evolutionary Way through Holistic Business-IT Delivery of Business Initiatives," In Proceedings of the 26th International Conference on Enterprise Information Systems, Volume 1, ISBN: 978-989-758-692-7, pp. 397-408, 2024.

[11] MACH Alliance, [Online], Available: https://machalliance.org, [retrieved: November, 2025].

[12] Gartner DXP Magic Quadrant, Available:https://www.gartner.com/reviews/market/digital-experience-platforms, [Online], [retrieved: November, 2025].

[13] MACH Alliance, [Online], Available: https://machalliance.org/insights-hub/composable-comes-of-age-in-the-gartner-dxp-magic-quadrant, [retrieved: November, 2025].

[14] ESICONF2025, [Online], Available: https://www.peernetwork.it/en/esiconf2025-agenda/, [retrieved: November, 2025].

[15] Global Logic Practitioner Perspective, Available: https://www.globallogic.com/wp-content/uploads/2023/04/Composable-Enterprise-PBC.pdf, [Online], [retrieved: November, 2025].

[16] Forbes, Available: https://www.forbes.com/councils/forbestechcouncil/2025/07/22/20-hidden-benefits-of-composable-architecture-in-enterprise-tech/, [Online], [retrieved: November, 2025].
[17] Contentstack, Available: https://www.contentstack.com/blog/composable/why-your-business-needs-a-packaged-business-capability-now, [Online], [retrieved: November, 2025].
[18] Value Innovation Labs, Available: https://valueinnovationlabs.com/blog/digital-transformation/why-composable-architecture-will-lead-digital-transformation-in-2025/, [Online], [retrieved: November, 2025].
[19] HCLSoftware, Available: https://www.hcl-software.com/blog/commerce/embracing-packaged-business-capabilities-pbcs-a-superior-middle-alternative-to-microservices-vs-monoliths, [Online], [retrieved: November, 2025].
[20] J. Barney, "Firm resources and sustained competitive advantage," In Journal of Management, Volume 17, Issue 1, pp 99-120, 1991.
[21] T. Offerman, C.J. Stettina, and A. Plaat, "Business capabilities: A systematic literature review and a research agenda," In International Conference on Engineering, Technology and Innovation (ICE/ITMC), pp. 383-393, 2017.
[22] J.L.G Dietz and H.B.F. Mulder, "Enterprise ontology: A human-centric approach to understanding the essence of organisation", ISBN:978-3-031-53361-7, Heidelberg: Springer, 2024.
[23] BIAN, [Online], Available: https://bian.org, [retrieved: November, 2025].
[24] NBility, [Online], Available: https://www.edsn.nl/nbility-model, [retrieved: November, 2025].
[25] LeadingPractise, [Online], Available: https://www.leadingpractice.com, [retrieved: November, 2025].
[26] SAFe Framework, [Online], Available: www.scaledagileframework.com, [retrieved: November, 2025].
[27] Object Management Group (OMG), [Online], Available: https://www.objectmanagementgroup.org/, [retrieved: November, 2025].
[28] The Open Group Architecture Framework (TOGAF), [Online], Available: https://www.opengroup.org/togaf, [retrieved: November, 2025].
[29] BIZBOK, [Online], Available: https://www.businessarchitectureguild.org, [retrieved: November, 2025].
[30] G. McGovern, "Top Tasks – A how-to guide", ISBN: 978-1-9164446-1-4, Silver Beach, 2018.
[31] R.C. Martin, "Clean architecture: a craftsman's guide to software structure and design", ISBN-13: 978-0-13-449416-6, Prentice Hall Press, 2017.
[32] E. Evans, "Domain-driven design: tackling complexity in the heart of software", ISBN-10: 0321125215, Addison-Wesley Professional, 2004.
[33] K. Gene, et al, "The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations", ISBN: 978-1942788003, IT Revolution, 2016
[34] H. Mannaert, T. Van Waes, and F. Hannes, "Toward a rejuvenation factory for software landscapes", Patterns 2024: The Sixteenth International Conference on Pervasive Patterns and Applications, April 14-18, 2024, Venice, Italy, 2024.